Report No. UIUCDCS-R-77-893 UILU-ENG 77 1747

AN INTRODUCTION TO PL/C

by

Douglas D. Dankel, II

September 21, 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-77-893

AN INTRODUCTION TO PL/C

BY

DOUGLAS D. DANKEL II

September 1977

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1 THE PROGRAM AND THE COMPUTER

After writing a program, the program must be input to the computer. Programs can be input to a computer on punched cards, paper tape, magnetic tape or from a teletype connected directly to the computer. Most beginning courses in computer science use punched cards. This chapter examines the way the cards should be punched so they can be properly interpreted by the computer and the ordering of the cards before reading them into the computer.

Once the program has been read into the computer, it is executed and the results of the execution are output. This output can occur as printed paper, punched cards, paper tape or magnetic tape. This chapter will also examine how the printed paper output is arranged because this form of output is most commonly used.

Those who are familiar with running jobs on the University of Illinois Computer System should skim this chapter for specifics concerning the use of the PL/C language before proceeding to Chapter 2. Others should read this chapter in detail.

1.1 THE PROGRAM INPUT -

Figure 1.1 shows the order of the cards in a PL/C program deck that must be used when running on the University of Illinois Computer System.

The JOB CARD is a green card that is placed on the top of

```
JOB CARD
ID CARDS
// EXEC PLC
$PL/C
     Program Statements
$DATA
     Data Cards
/*
```

FIGURE 1.1 - THE PL/C PROGRAM CARD DECK ARRANGEMENT

every program deck before reading it into the computer. A stack of JOB CARDs is found next to all card readers. Whenever a job is read into the machine, place a new JOB CARD on the top of the deck. It notifies the computer that what follows is a job to be run.

The ID cards provide information to the operating system about the job. Normally two ID cards are used. On the first is specified the user's name and the number of the account which should be charged for the job. The second card has the code word associated with the specified account number, what system the job should be run on (HASP, EXPRESS, etc.) and what computer resources should be allocated for the job (number of lines, time, amount of storage, etc.). Further information on the ID cards can be obtained by reading the CSO REFERENCE GUIDE S01.N01 on ID cards. Two possible ID cards are given in Figure 1.2. ID cards are given to the students by the T.A. or grader at the beginning of the semester. A copy of the ID cards should be made and stored in a safe place in case the originals become lost or damaged. If the ID cards are lost, notify the T.A. immediately so a new set of ID cards can be issued.

The // EXEC PLC card states that the program should be executed using the PL/C compiler. Care must be taken in punching the ID cards and the EXEC card to make sure that they are punched exactly as shown. They should start in column 1 on the card with

only one blank used where blanks are shown. No blanks should be

```
/*ID PS=(6974,5001000),NAME='DANKEL'
/*ID CODE=DQR,SYSTEM=EXPRESS,LINES=350
```

FIGURE 1.2 - SAMPLE ID CARDS

added in other places because they will result in the program not being executed or executed without the requested computer resources.

All cards to this point have been information to the operating system, telling it what should be done with the program. The $PL/C card tells the PL/C compiler that what follows is the start of a PL/C program. Several options are available when the program is run and if any of these is desired it must be specified on this card. Options that are available are listed in Appendix A. The statements of the actual program follow in the order that the programmer wishes them to be executed.

The $DATA card tells the compiler that the program has ended and data cards follow. The data cards follow the $DATA card in the order that they will be read by the computer.

The /* card tells the operating system that this is the end of this program and that is should now get ready for the next job.

1.2 CARD FORMATS -

For PL/C jobs run on the University of Illinois Computer System the following card formats must be observed:

1) All ID and operating system cards must start in column 1. These cards are easily identifiable because they start with /* or //.

2) PL/C special cards must also start in column 1. These cards can be identified by their starting $.

3) The actual PL/C program cards must be started after column 1 and must not continue beyond column 72. Columns 73-80 are reserved for the numbering of the cards in case the cards should accidentally be dropped and their order disturbed.

4) Data cards can use all 80 cclumns of the card. Care should be taken to ensure that data cards do not contain /* or // in columns 1 and 2.

Figure 1.3 shows a sample program as it might appear before being read into the computer.

1.3 PROGRAM OUTPUT -

Programs run on the University of Illinois Computer System will always has the following information printed in the program output:

1) Header - Starts with the program's job number, programmer's name and the time at which the job was run being printed out on six consecutive lines. This is done to easily identify the start of each person's output. Following this is several lines giving the computer resources requested by the program. For example, the amount of time it has requested to use, the maximum number of lines it will print, what system it should run under (EXPRESS, HASP, etc.) and the type of computer language that the program is written in. This is produced by the operating system which controls what the computer does.

2) Source Listing - Consists of a listing of the statements of the program and any error

```
//J9991318 JOB
/*ID PS=(6974,5001000),NAME='DANKEL'
/*ID CODE=DQR,SYSTEM=EXPRESS,LINES=350
// EXEC PLC
$PL/C
   /* CS 121 MP 0                                          */
   /* PROGRAM TO PRINT A TABLE OF SQUARES                  */
   /* COMMENTS SUCH AS THIS AND THE ONES ABOVE ARE         */
   /* IGNORED BY THE COMPUTER                              */

   TABSQ:PROCEDURE OPTIONS(MAIN);
    DCL (K,M,N) FIXED DEC;

   /* READ NUMBER OF SQUARES DESIRED                       */

    GET LIST(N);

   /* PRINT HEADINGS                                       */

    PUT LIST('TABLE OF SQUARES UP TO',N);
    PUT SKIP(2) LIST('NUMBER','SQUARE');
    M=0;

   /* REPEAT NEXT STATEMENTS N TIMES                       */

    DO WHILE(M<N);

      M=M+1;
      K=M*M;
      PUT SKIP LIST(M,K);
      END;

    END TABSQ;
$DATA
32
/*
```

FIGURE 1.3 - A SAMPLE PROGRAM

or warning messages that they might have caused.

3) Cross Reference and Attribute Listings - If the program is written in PL/C or PL/I a listing of all the identifiers, where they are used in the program, and what type of identifiers they are will be given at this point. PL/I will always give this listing unless specifically requested not to give it. PL/C on the other hand, will not give this listing unless asked to. (See Appendix A).

4) Execution Output - Any output specified by the program is printed at this point. The output is always started at the top of a new page.

5) Post-mortem Dump - If the program is run under PL/C a post-mortem dump will be given. This dump consists of the following:
A) Variables used in the program and their values when the program halted.
B) All labels used, what statement they are used in, and the number of times they were encountered when the program was executed.
C) A dynamic flow trace showing the last 18 transfers that occurred during execution of the program - these transfers are points in the program where the statements were not executed in numerical order.
D) The total amount of storage used by the program while it was executing and how much remained available at the end of execution.
E) The amount of time used to compile (translate) the program from PL/C into the machine's language, the time that was used for execution, and the total time that the program used the computer.

6) Trailer - Starts with a six line printing of the programmer's job number, programmer's name, and the time that the program was executed. Also listed are the turn around time (time from when the job was read into the machine until it was printed), the total

amount of computer resources that were used by the program, the cost of the job, and how much money is left in the user's account.

Figure 1.4 shows the output that results from running the program shown in Figure 1.3.

```
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START
********* J9991318 ***** 437 ************** DANKEL                      ************** 10.03.55 AM * 26 DEC 75 ********** START


******************************************** H A S P  (3.1)  S Y S T E M  L O G **********************************************


$*10.03.42 JOB 437 RELEASED
$ 10.03.42 JOB 437 -- J9991318 -- IN EXEC - INIT 6 - CLASS X
$ 10.03.45 JOB 437 J9991318 END EXECUTION.
$ 10.03.51 JOB 437 J9991318 ON PRINTER1


******************************* O S / M V T  (21.7)  S Y S T E M  M E S S A G E  B L O C K ********************************


************************************* UNIVERSITY OF ILLINOIS EXPRESS SYSTEM VERSION 2.0 *************************************


  //J9991318 JOB (6974),TIME=(0000,10),MSGLEVEL=(2,0)                              JOB 437
  //*ID PS=(6974,50C100G),NAME='DANKEL',EJECT=YES,HOLD=YES
  //*ID,SYSTEM=EXPRESS,LINES=350
  // EXEC PLC

XPR0001*******TIME AT INIT PL/C      IS 10.03.43.13
```

FIGURE 1.4 - Output from the Sample Program

```
$PL/C

*OPTIONS IN EFFECT*     TIME=(0,10),PAGES=20,LINES=2000,NOATR,NOXREF,FLAGW,NOBNDRY,NOCMNTS,SOURMGIN=(2,72,1),ERRORS=(50,50),
*OPTIONS IN EFFECT*     TABSIZE=10212,UDEF,SOURCE,OPLIST,NOCMPRS,NOHDRPG,AUXIO=10000,LINECT=60,NOALIST,MCALL,MTEXT,DUMP=(S,F,L,E,U,
*OPTIONS IN EFFECT*     R),DUMPE=(S,F,L,E,U,R),DUMPT=(S,F,L,E,U,R)

 /* CS 121 MP D                                   */                    PL/C-R7.1--DD 12/26/75 10:03 PAGE   1

  STMT LEVEL NEST BLOCK MLVL  SOURCE TEXT

                              /* CS 121 MP D                                   */
                              /* PROGRAM TO PRINT A TABLE OF SQUARES           */
                              /* COMMENTS SUCH AS THIS AND THE ONES ABOVE ARE  */
                              /* IGNORED BY THE COMPUTER                       */

     1                        TABSQ:PROCEDURE OPTIONS(MAIN);
     2    1          1          DCL (K,M,N) FIXED DEC;

                              /* READ NUMBER OF SQUARES DESIRED                */

     3    1          1          GET LIST(N);

                              /* PRINT HEADINGS                                */

     4    1          1          PUT LIST('TABLE OF SQUARES UP TO',N);
     5    1          1          PUT SKIP(2) LIST('NUMBER','SQUARE');
     6    1          1          M=0;

                              /* REPEAT NEXT 5 STATEMENTS N TIMES */

     7    1          1          DO WHILE(M<N);

     8    1    1     1            M=M+1;
     9    1    1     1            K=M*M;
    10    1    1     1            PUT SKIP LIST(M,K);
    11    1    1     1            END;

    12    1          1          END TABSQ;

ERRORS/WARNINGS DETECTED DURING CODE GENERATION:

      WARNING: NO FILE SPECIFIED. SYSIN/SYSPRINT ASSUMED. (CGOC)
```

FIGURE 1.4 - Continued

```
TABLE OF SQUARES UP TO        32

 NUMBER          SQUARE
      1               1
      2               4
      3               9
      4              16
      5              25
      6              36
      7              49
      8              64
      9              81
     10             100
     11             121
     12             144
     13             169
     14             196
     15             225
     16             256
     17             289
     18             324
     19             361
     20             400
     21             441
     22             484
     23             529
     24             576
     25             625
     26             676
     27             729
     28             784
     29             841
     30             900
     31             961
     32            1024

IN STMT   12  PROGRAM RETURNS FROM MAIN PROCEDURE.
```

FIGURE 1.4 - Continued

```
/* CS 121 MP 0                                                    */          PL/C-R7   POST-MORTEM DUMP   PAGE   1

IN STMT   12  SCALARS AND BLOCK-TRACE:

***** MAIN PROCEDURE TABSQ

N=      32              M=      32              K=    1024


NON-C PROCEDURE EXECUTION COUNTS:
NAME        STMT COUNT  NAME        STMT COUNT  NAME        STMT COUNT  NAME        STMT COUNT
TABSQ       0001 00001
```

COMPILATION STATISTICS (0012 STATEMENTS)						
SECONDS	ERRORS	WARNINGS	PAGES	LINES	CARDS	INCL'S
.25	0	1	1	45	30	0

EXECUTION STATISTICS						
SECONDS	ERRORS	WARNINGS	PAGES	LINES	CARDS	INCL'S	AUX I/O
.20	0	0	1	39	1	0	0

BYTES	SYMBOL TABLE	INTERMEDIATE CODE	OBJECT CODE	STATIC CORE	AUTOMATIC CORE	DYNAMIC CORE	TOTAL STORAGE
USED	558(1K)	334(1K)	372(1K)	342(1K)	230(1K)	0(0K)	1494(2K)
UNUSED	40290(39K)	40310(39K)	80990(79K)	80430(78K)	80430(78K)	40310(39K)	80430(78K)

```
XPR0001*******STEP TIME IS    .64 SECS
```

FIGURE 1.4 - Continued

```
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.05 AM * 26 DEC 75 ********** END


JOB CLASS: X       MAGIC NO: 3*SECS + .05*IOREQ + .01*K*K=     1       TURNAROUND:    33 MINUTES

PS: 6974

          1 COVER CHARGE--XPRESS BATCH
         37 CARDS READ--CSO NORTH
         37 CENTISECONDS--360/75 PROCESSOR TIME
         16 INPUT/OUTPUT REQUESTS--360/75
          3 UNITS CHARGED--360/75 EXECUTION
         29 UNUSED PRINT LINES CHARGED
        135 LINES PRINTED--CSO NORTH


THIS RUN USED         .54 SERVICE UNITS     (ONE SERVICE UNIT EQUALS $1.00)

USER 500100D HAS        145.88 SERVICE UNITS REMAINING

PS       6974 HAS        160.75 SERVICE UNITS REMAINING







********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
********* J9991318 ***** 437 ************** DANKEL                ************** 10.04.07 AM * 26 DEC 75 ********** END
```

FIGURE 1.4 - Continued

CHAPTER 2 THE PROGRAM AND ITS STATEMENTS

> "The computer is like a malevolent genius. If you don't tell it what to do, it won't do it."

This chapter examines the form in which programs should be written and the different statements that can be used in PL/C. The use of each statement will be explained with examples. For further details and examples consult the IBM System 360 PL/I (F) Language Reference Manual Form C28-2801, IBM System 360 OS PL/I-F Programmer's Guide Form C28-6594, and AN INTRODUCTION TO PROGRAMMING, A STRUCTURED APPROACH USING PL/I AND PL/C-7 by Richard Conway and David Gries.

2.1 GENERAL PROGRAM FORMAT -

The statements that compose an actual program must be placed between the $PL/C and $DATA cards. These program statements define where the program starts and ends. Contained between these two statements are executable and non-executable statements. After a program has been read into a computer, the computer first translates or compiles the program into a form that it can understand. Executable statements are compiled into actual

machine instructions which the computer will execute. Non-executable statements are used to provide information to the computer when it compiles the program. Figure 2.1 shows how the

```
$PL/C
  NAME: PROCEDURE OPTIONS(MAIN);
     Non-executable Statements
     Executable Statements
     END NAME;
$DATA
```

FIGURE 2.1 - GENERAL PROGRAM FORMAT

program is arranged in relation to the PL/C and $DATA cards. Details of the PROCEDURE statement and the END statement are given later in this chapter.

Please note that the statements of the program must be punched in columns 2 thru 72. Card column 1 is reserved for carriage control characters. These characters are -

blank	space one line before printing
0	space two lines before printing
-	space three lines before printing
+	do not space before printing (overprint)
1	eject a page before printing

These characters will not be printed on the output when used. Any other characters occurring in column 1 are assumed to have been accidentally punched beginning in column 1 instead of column 2 and PL/C will generate an error to identify this.

Each statement written in PL/C is ended with a semicolon, so more than one statement can occur on a given card. This practice is not recommended, however, because the resulting program is difficult to read.

If a statement is too long and cannot fit on one card, it can be continued on the next card. Because no semicolon is found on the first card, PL/C will assume the statement is continued on

the next card. Care must be taken to insure that identifiers and character strings are not broken across cards; otherwise, statements can be broken at any point.

2.2 IDENTIFIERS -

Identifiers are labels or variables used within a program. Labels are used to identify particular statements in the program. Variables are used to store values that the program is using. Identifiers can contain a maximum of 31 alphanumeric (alphabetic and numeric) characters. They must start with an alphabetic character (the letters 'A' to 'Z' and '$', '#' or '@') and can include the break character, '_'.

```
ALLOCATE  BEGIN  BY  CALL CHECK CLOSE DECLARE
(DCL) DO ELSE END ENTRY EXIT FLOW FORMAT FREE
GO  GOTO  IF  NO  NOCHECK  NOFLOW  ON  OPEN
PROCEDURE  (PROC)  PUT  READ  RETURN  REVERT
SIGNAL STOP THEN TO WHILE WRITE
```

FIGURE 2.2 - KEYWORDS IN PL/C

Keywords are identifiers which, when used in a special way, have a special meaning. Keywords cannot be used as labels or variables in PL/C. A list of the keywords is given in Figure 2.2 with all allowed abbreviations for some of the keywords following in parentheses.

Several legal and illegal identifiers are given in Figure 2.3 and 2.4. Before continuing verify that these identifiers are legal or illegal.

```
#M    GETT    I23    QA4_73    CNT1_FOR_DOGS    $
@HOME    AQW12##    $S___A4    RESTER
```

FIGURE 2.3 - LEGAL IDENTIFIERS

```
8A      _A      A+B      A//BCD
A_VERY_VERY_VERY_LONG_VARIABLE_NAME
```

FIGURE 2.4 - ILLEGAL IDENTIFIERS

2.3 ASSIGNMENT STATEMENT -

Assignment statements are used to perform calculations and to assign the result of the calculations to variables. The general form of an assignment statement is shown in Figure 2.5. '{' and '}' are used in Figure 2.5 and subsequent figures to denote something that can be repeated zero or more times.

```
VARIABLE { , VARIABLE } = EXPRESSION ;
```

FIGURE 2.5 - GENERAL FORM OF AN ASSIGNMENT STATEMENT

The execution of an assignment statement proceeds as follows: First, the values of the variables used in the expression are determined. The expression is then evaluated with these values to obtain the result. Finally, the resulting value is assigned to each of the variables specified on the left of the equal sign.

Suppose that an assignment statement was desired for the following equation:

$$A = \frac{B + C^{2(Q)}}{D - E(F)} \qquad (2.1)$$

In examining a keypunch machine it will be immediately noticed that equations can only be punched in a linear fashion; there is no way to punch exponentiation or division because they require expressions to be written on more than one line. Because of this, special symbols are used to represent the mathematical operators. These operators are given in Figure 2.6.

SYMBOL	ARITHMETIC OPERATION
+	ADDITION
-	SUBTRACTION
*	MULTIPLICATION
/	DIVISION
**	EXPONENTIATION

FIGURE 2.6 - PL/C ARITHMETIC OPERATORS

Using the operators given in Figure 2.6, one might write the above equation as follows

```
A = B + C ** 2 * Q / D - E * F ;                    (2.2)
```

However, a problem arises. Does assignment statement (2.2) mean

$$A = B + \frac{C^{2}}{D}Q - E(F) \qquad (2.3)$$

or the original equation (2.1)? In actuality, what the assignment statement (2.2) means is the second equation (2.3). This is due to a priority that is assigned to each of the operators. This

priority scheme is given in Figure 2.7. When assignment

```
EXPONENTIATION
MULTIPLICATION AND DIVISION
ADDITION AND SUBTRACTION
```

FIGURE 2.7 - PRIORITY SCHEME FOR THE OPERATORS GIVEN FROM HIGHEST TO LOWEST PRIORITY

statements are evaluated, the priority scheme is used to determine the order in which the operations specified will be carried out. All exponentiation operations specified in an assignment are executed before any other operations because they have the highest priority. All multiplication and divisions are performed and, finally, additions and subtractions are performed. Notice that this solves most problems in interpreting assignment statements, but some problems still remain. For example, does

```
A / B * C                                    (2.4)
```

mean

```
 A
--- * C                                     (2.5)
 B
```

or

```
  A
-----                                        (2.6)
B * C
```

and does

```
A ** B ** C                                  (2.7)
```

mean

$$A^{(B^{C})} \tag{2.8}$$

or

$$(A^{B})^{C} \tag{2.9}$$

The following rule applies to two or more operations of equal priority that occur consecutively:

> If two or more operations of equal priority occur consecutively, the operations are to be performed in order from left to right, except if the operations are exponentiations, in which case the operations should be performed in order from right to left.

Applying this rule to equations (2.4) and (2.7) it is found that they will be interpreted as equations (2.5) and (2.8), respectfully.

At anytime the priority scheme and ordering rule may be changed by the use of parentheses. All operations inside a set of parentheses are executed using the priority scheme and ordering rule before the other operations so that the proper grouping of values can be acccmplished.

Using these facts, the following assignment statement is developed to express equation (2.1)

```
A = (B + C ** (2 * Q)) / (D - E * F);          (2.10)
```

Before continuing verify that this assignment statement is correct.

As shown in the general form for assignment statements, more than one variable can be assigned the value computed by the expression. These variables should be placed to the left of the equal sign and should be separated by commas. For example, if A, P and Z were all to obtain the value of the expression in the assignment statement (2.10) then the statement could be written

```
A, P, Z = (B + C ** (2 * Q)) / (D - E * F);    (2.11)
```

2.4 COMMENTS -

Comments are non-executable statements. They are used to provide information to the programmer and others who might look at the program about what the program does. There is nothing worse than picking up a program written a year or two earlier and trying to figure out what it does when there are no comments. Comments aid the programmer when he has not looked at the program in a long time and also aid someone else who is looking at the program. Comments should explain in general what the program does and any sections of the program which are unclear. Very few programs can be considered to be over commented so if in doubt put in a comment. Students have a very bad habit of waiting until the last moment to add their comments. Do not wait! Put in comments when the program is originally written and still fresh in your mind.

Comments are started with a /* and are ended with a */ . They can occur anywhere in a program except in an identifier or character string. Figure 2.8 shows the legal use of a comment while Figure 2.9 shows the same statement with a comment illegally used.

```
X = FIRST /* A LEGAL COMMENT */ + SECOND;
```

FIGURE 2.8 - LEGAL USE OF A COMMENT

```
X = FI/* AN ILLEGAL COMMENT */RST + SECOND;
```

FIGURE 2.9 - ILLEGAL USE OF A COMMENT

Remember that /* also signifies the end of the program if punched in columns 1 and 2, so take care not to punch comments starting in column 1.

Remember also to end comments with a */ before column 73 on the card. If a comment is not ended or is 'ended' beyond column 72, the statements in the program following the comment will be treated as though they are part of the comment. This will continue until the end of the next comment is found.

2.5 DECLARE STATEMENT -

The DECLARE statement is a non-executable statement used to

```
DECLARE IDENTIFIER ATTRIBUTES;
```

FIGURE 2.10 - GENERAL FORM OF A DECLARE STATEMENT

specify what identifiers will be used in the procedure and what type of identifiers they are. The DECLARE statement provides this information to the compiler so it can set aside space for these variables. DECLARE statements may be placed anywhere in a procedure but are usually placed at the beginning of the procedure so they are not confused with or 'lost' in the exeutable statements.

Figure 2.10 shows the general form of the DECLARE statement. IDENTIFIER is the identifier being declared and the ATTRIBUTES specify what type the identifier is. The following two DECLARE statements define a variable X which contains a fixed decimal number and DOG which contains a float binary number:

```
DECLARE X FIXED DECIMAL;
DECLARE DOG FLOAT BINARY;
```

More than one identifier can be declared in a DECLARE statement for example:

```
DECLARE ( X, DOG, PETE) FLOAT DECIMAL,
( HARRY,MIKE) FIXED DECIMAL;
```

Note the use of parentheses to apply attributes to several variables and the use of the commas to allow more variables to be declared. An alternative way of writing the above DECLARE statement is:

```
DECLARE (( X, DOG, PETE) FLOAT,
( HARRY, MIKE) FIXED ) DECIMAL;
```

where parentheses are used to apply the attribute DECIMAL to both groups of variables being declared.

Because every identifier must have some attributes, default values are assumed when an identifier is not declared or is declared with no attributes given. These default values are assigned according to the first letter in the identifier and are-

> If the first letter is I, J, K, L, M or N then the identifier's attributes are FIXED BINARY; otherwise, FLOAT DECIMAL are used.

The following is a list of the commonly used attributes. For more detail and a more complete list consult AN INTRODUCTION TO PROGRAMMING by Conway and Gries or the IBM System 360 PL/I Reference Manual.

1) Type Attributes - specify the type of information that the identifiers being declared can contain.

A) Arithmetic Variables -

1) Base - can be specified as either DECIMAL or BINARY. This refers to how the values of the variables are represented within the computer. Note that binary arithmetic is two to three times faster on an IBM 360 than decimal arithmetic and binary and decimal are about equal on an IBM 370. For this reason, try to declare commonly used arithmetic variables BINARY in type.

2) Scale - can be specified as either FLOAT or FIXED. Variables declared FLOAT are stored in scientific notation - the position of the decimal point is allowed to 'float' in the number so a maximum number of significant digits of the value are kept. FIXED variables have the decimal point fixed to a specific position.

3) Precision - specifies the number of digits to be used in representing the number. If the precision is not specified the following default values will be used:

```
FIXED DECIMAL (5,0)
FIXED BINARY  (15,0)
FLOAT DECIMAL (6)
FLOAT BINARY  (21)
```

The first number is the maximum number of digits in the variable's representation within the machine, while the second number, if given, is the number of those digits that are fractional. The maximum number of digits that may be specified is 15 for FIXED DECIMAL, 31 for FIXED BINARY, 16 for FLOAT DECIMAL, and 53 for FLOAT BINARY. Normally a precision is not specified for arithmetic variables, the default values are used. The range of values that variables an have under the various declarations with default precisions is

-99999 to 99999 for FIXED DECIMAL

-32767 to 32767 for FIXED BINARY

approximately ±10 **-78 to ±10***+75 for FLOAT DECIMAL

approximately ±2**-260 to ±10***252
for FLOAT BINARY

B) Character String Variables - (see Chapter 5 for additional information)

1) Base - must be specified as CHARACTER.

2) Scale - can be specified as VARYING if the variable is to hold a varying number of characters.

3) Precision - specifies the number of characters that the variable can hold. A character string variable declared as VARYING can contain from zero to this maximum number of characters. Variables not specified as VARYING will always contain this number of characters. A maximum value of 256 can be used in PL/C. This must be specified when declaring character string variables.

C) Bit String Variables -

1) Base - must be specified as BIT.

2) Scale - can be specified as VARYING if the variable is to hold a varying number of bits.

3) Precision - specifies the number of bits that the variable can hold. A bit string variable declared as VARYING can contain from zero to this maximum number of bits. Variables not specified as VARYING will always contain this number of bits. A maximum value of 256 can be used in PL/C. This must be specified when declaring bit string variables.

D) Label Variables - the only specification for label variables is the base LABEL.

2) Initial Attributes - this can be used to give initial or starting values to variables. Only constants not expressions can be specified. Simple variables and arrays cannot be initialized together. The initial values are assigned to the variables only once, when the program in compiled.

Array initial values are assigned in row-major order (the rightmost subscript is varied the fastest). If not enough values are given, the remaining array elements are not initialized, while if more values are given than array elements, the extra initial values are ignored.

```
DECLARE (I FIXED BINARY,
        (J,K) FIXED DECIMAL) INITIAL(15);
DECLARE CAT(5) FLOAT DECIMAL(2)
        INITIAL(1,2,3,4,5,6);
DECLARE DOGS(5) CHARACTER(10) INIT((3)('PUPPY'));
DECLARE MICE CHARACTER(10) VARYING;
DECLARE (A,B) FIXED DECIMAL INIT(5,10);
```

FIGURE 2.11 - SAMPLE DECLARATION STATEMENTS

Figure 2.11 contains several DECLARE statements. In the first declaration the identifier I is declared to be a FIXED BINARY variable, while J and X are declared to be FIXED DECIMAL variables. I, J, and K are all initialized to a value of 15. An array called CAT of FLOAT DECIMAL variables is declared in the second declaration statement. Each element of the array can contain a FLOAT DECIMAL number with a maximum of two significant digits. Notice that one more value is given in the INITIAL specification than there are places in the array, so the last value in the initialization specification is not used. The third declaration statement declares an array called DOGS which is

CHARACTER in type. Because VARYING is not specified, each element of the array will contain ten characters. Notice the replication factor used in the initialization specification. It causes the first three elements of the array to receive the characters 'PUPPY' followed by five blanks. The remaining two elements in the array are given no initial value. The next declaration is for the variable MICE which can contain from zero to ten characters. No initial value is given to MICE, so its value at the start of the execution of the program is undefined. The last declaration statement shows a common programming error. The programmer intended to declare A with an initial value of 5 and B with an initial value of 10. But because A and B are declared together, they both receive a starting value of 5. The initial value of 10 is ignored.

2.6 DO STATEMENT

The DO statement is used to specify that a group of statements is to be executed several times. Corresponding to every DO statement is an END statement which specifies the limits of the statements to be executed repetitively. The three basic forms of the DO statement are examined below.

2.6.1 NONITERATIVE DO

The noniterative or simple DO statement is the most elementary form of the DO. It causes all of the statements up to the END statement to be executed once and only once. Figure 2.12 shows how this statement might appear in a program.

This form allows all statements between the DO and the END statements to be counted as though they were one statement. Normally this statement is used when more than one statement needs to be executed in a THEN or ELSE clause of an IF statement. For more information see section 2.9 on IF statements.

```
        :
        :
        DO;
          :
          :
          END;
        :
        :
```

FIGURE 2.12 - GENERAL FORM OF THE NONITERATIVE DO STATEMENT

2.6.2 ITERATIVE DO

The iterative DO is used to repeat a group of statements a specific number of times. Figure 2.13 shows the general form of this DO statement.

```
DO VAR=STRT TO STP BY INCR;
        :
        :
      END;
```

FIGURE 2.13 - GENERAL FORM OF THE ITERATIVE DO STATEMENT

STRT is the initial or starting value given to variable VAR, STP is the final or stopping value,and INCR is the incremental amount. STRT, STP and INCR can be either constants or expressions. The iterative DO statement is executed in the

following manner. First, the values of STRT, STP and INCR are determined. Once these values have been determined, they cannot change. The variable, VAR, is assigned the value of STRT. A check is made to see if the incremental value is positive or negative. If positive, a test is made to make sure that the value of VAR is not greater than the stopping value; if the incremental value is negative, the test is made for the value of VAR being not less than the stopping value. If the test is true, the statements of the loop are executed. When the END statement is encountered, the value of the variable is incremented by INCR and the test is performed again. This process is continued until the test fails. The computer will then start executing the statements immediately following the END statement of the loop. Examine the sample program segment in Figure 2.14. The iterative loop will cause the

```
  :
  :
A=0;
DO I=1 TO 5 BY 2;
  X:A=A+I;
  END;
  :
  :
```

FIGURE 2.14 - SAMPLE ITERATIVE DO LOOP WITH CONSTANT VALUES

statement labeled X to be executed for values of I equal to 1, 3, and 5. The value that I will have when the looping stops is 7. Verify that these values are correct. What is the value that A will have after execution?

Figure 2.15 shows the same basic iterative DO loop with an expression for the TO clause. Remember that the initial, stopping and incremental expressions are evaluated before the loop is executed and remain constant for as long as the loop is executing. The TO clause is then evaluated to 11 and that value remains until the loop is exited. This causes the loop to be executed for values of I equal to 1, 4, 7, and 10. The final value for I is 13 and the statements of the loop are not executed with this value. Again verify that these values are correct and determine the value that A will have after execution of the

statements.

```
  :
  :
A=10;
DO I=1 TO A+1 BY +3;
  X:A=A+I;
  END;
  :
  :
```

FIGURE 2.15 - SAMPLE ITERATIVE DC LOOP WITH EXPRESSIONS

```
  :
  :
A=10;
DO I=A TO 1 BY -2;
  X:A=A+I;
  END;
  :
  :
```

FIGURE 2.16 - SAMPLE ITERATIVE DO LOOP WITH A NEGATIVE INCREMENT

Figure 2.16 illustrates an iterative DO loop with a negative increment. Because the increment is negative the test is made for the value of I being less than the TO clause value of 1. The loop is executed for values of I equal to 10, 8, 6, 4, and 2.

The iterative DO loop can also be written without specifying a TO or BY clause. If the BY clause is not specified, it is assumed to be +1. Figure 2.17 illustrates such a DO loop.

Because a BY clause is not given it is assumed to be +1 and

the statement labeled X will be executed 6 times. By not

```
  :
  :
A=0;
DO I=5 TO 10;
  X:A=A+1;
  END;
  :
  :
```

FIGURE 2.17 - SAMPLE ITERATIVE DO LOOP WITHOUT A BY CLAUSE

specifying a TO clause, an infinite loop can be created. Figure

```
  :
  :
A=0;
DO I=1 BY 2;
  X:A=A+I;
  END;
  :
  :
```

FIGURE 2.18 - SAMPLE ITERATIVE DO LOOP WITHOUT A TO CLAUSE

2.18 illustrates an infinite loop. Each time the END statement is encountered, I is incremented by 2 and because no TO clause is specified, the statement labeled X is then executed with no stopping test being made. This loop will continue to be executed until the job containing it exceeds its time limit.

Note that whenever the value of the variable exceeds the TO clause value, execution of the statements in the loop will stop. So if the initial or starting value exceeds the TO clause the statements of the loop will not be executed even once.

2.6.3 DO WHILE

The DO WHILE statement is used to repeat a group of statements like the iterative DO but is in many ways simpler than the iterative DO statement. Figure 2.19 shows the general form for the DO WHILE statement. Notice that only a condition is specified in the DO statement. This CONDITION is evaluated, resulting in either a true or false value. If the condition is true, the statements between the DO and END statements will be executed. The condition is then retested and if it is still true

```
   :
   :
DO WHILE(CONDITION);
    :
    :
   END;
   :
   :
```

FIGURE 2.19 - GENERAL FORM OF THE DO WHILE STATEMENT

the statements within the loop are again executed. This process is repeated until the condition becomes false. The looping is then stopped and execution continues with the first statement following the END statement for the loop. The condition is tested upon the initial entry to the loop and every time the loop is to be repeated (whenever the END statement is encountered).

Figure 2.20 shows a sample DO WHILE loop. Note the use of the PL/I logical OR symbol (|) to form compound condition. As long as either I<=5 or A=-1 the loop will continue to be executed. When the condition is initially tested, it is found to be true because I is less than 5. The statements of the loop are executed resulting in I=2 and A=0. The condition is retested and found to be true because I is less than 5. The statements of the loop are executed resulting in I=4 and A=-1. The condition is

tested and found to still be true because I is less than 5 and A

```
 :
 :
I=0;
A=1;
DO WHILE (I<=5 | A=-1);
  I=I+2;
  A=A-1;
  END;
 :
 :
```

FIGURE 2.20 - SAMPLE DO WHILE STATEMENT

is equal to -1. Again the statements in the loop are executed resulting in I=6 and A=-2. When the condition is now tested it is found to be false. The loop is exited and execution continues with the statement following the END statement.

2.7 END STATEMENT

The END statement is used to end PROCEDUREs and DO groups. Figure 2.21 shows the general form of the END statement. LABEL is

```
END { LABEL } ;
```

FIGURE 2.21 - GENERAL FORM OF AN END STATEMENT

the statement label given to the PROCEDURE or DO statement which matches this END statement. If the label is used, a check is made to make certain that the END statement truly does correspond to the DO or PROCEDURE statement with that label. If that DO group

or PROCEDURE has already been ended, an error message will be given and the label removed from the END statement. If the END statement occurs too early (i.e., other DO groups or PROCEDUREs have not yet been ended), PL/C and PL/I will insert the needed END statements without an error or warning message. For this reason, it is not recommended that the labels be used on END statements. Make certain that END statements occur in the right places and that none are left out, otherwise the logic of the program could be completely changed.

2.8 GET AND PUT STATEMENTS

One important feature of all computer languages is the ability to GET (read) and PUT (write) information. Without these statements a program cannot communicate with the outside world, obtain values to use in the calculations or return the values that have been computed. Several forms of the statements exist and examination will start with the various forms of the GET statement.

2.8.1 GET STATEMENT

The GET statement is used for reading information from the data provided to the program. Figure 2.22 shows the various forms of the GET statement. Several OPTIONs are available to be used. Any, all or none of these OPTIONs can be used in any of the forms of the GET statement. These OPTIONs are:

1) FILE(FILE_NAME) - This allows the programmer to specify the place where the computer should go to read the information requested in this statement. Normally, the programmer wishes to read from cards following the $DATA card so the FILE_NAME used is SYSIN resulting in

FILE(SYSIN)

If no file is specified the computer assumes that

SYSIN is the file that should be used for input, so

```
FORM 1 - GET [OPTIONS] LIST ( VARIABLE
         [,VARIABLE] );

FORM 2 - GET [OPTIONS] EDIT ( VARIABLE
         [,VARIABLE] )( FORMAT );

FORM 3 - GET [OPTIONS] DATA [ ( VARIABLE
         [,VARIABLE] ) ] ;

FORM 4 - GET [FILE(FILE_NAME)] SKIP[(EXP)] ;
```

FIGURE 2.22 - VARIOUS FORMS OF THE GET STATEMENT

this option need not be specified.

2) COPY - This causes all values read to be output on the line printer in the form that they are read. This option is quite useful in debugging to check that the information is being read in correctly.

3) SKIP [(EXP)] - This causes the computer to ignore the rest of the information on the current card and to go to the start of the next card before reading information. If (EXP) is used, the value of the expression is determined as an integer and that many cards are 'skipped' in the input file. If the value of the expression is <1, the value is set to 1. Note that the first SKIP results in the remainder of the current card being skipped. Even if only one column on the card has not be read. Further skips will cause whole cards to be skipped.

The GET statement must always start with the keyword GET. The option, if any, and the LIST group, EDIT group or DATA group can occur in any order.

LIST, EDIT, and DATA each specify the way the data appears in the input file. LIST means that the data items are in a list.

Each data item is separated from the next by blanks and/or a comma. The computer assigns the first data item to the first variable given inside the parentheses, the second data item to the second variable, etc. until it runs out of variables in the list. When the next GET statement is encountered, the process starts again where the first GET statement left off in the data items. Figure 2.23 shows a possible GET LIST statement. When this

```
GET SKIP(2) LIST (A, B, C) COPY;
```

FIGURE 2.23 - SAMPLE GET LIST STATEMENT

statement is executed, it will cause the computer to skip what remains on the current data card, skip all of the next card, and then read values for A, B, and C, outputting the values, in the form that they were read, on the printer.

The GET EDIT statement also has a list of variables to be read. The variables are specified exactly like the GET LIST variables. Following the list of variables is an additional list which specifies how the data values should be read from the cards. Specified below is a list of all format specifications that can be used.

F(W{,D}) - a number is to be read. It should be read as a fixed decimal number with a total width of W columns on the card. If a value for D is given, it specifies the number of digits to the right of the decimal point in the number. The width includes a column for the sign and a column for the decimal point if they are given in the number.

E(W,D) - a number should be read that is possibly in exponential form (scientific notation). It has a total width of W columns with D columns to the right of the decimal point. W includes columns for a sign, decimal point, the exponential E, a sign on the exponent, and two columns for the exponent's value.

A(W) - a character string is to be read. It contains a total of W columns or characters.

X(W) - this means to skip W columns in the input. If anything is punched in those columns, it will not be read.

SKIP[(EXP)] - this has the same effect as the skip as an option.

COL(EXP) - the expression is evaluated and the computer goes to that column on the current card. If that column has already been passed on the current card, it goes to the specified column on the next card.

Note that the first three format specifications given tell how to read the information that is on the cards, while the last three specifications are used to get the computer to the proper location on the card to read the values. Suppose the data cards

```
00000000011111111112222222223333333333.....8
12345678901234567890123456789012345678 9     0

  23.4E14ABC478.36923418   56734.56543
        423       78
937   14987
```

FIGURE 2.24 - SAMPLE DATA CARDS

shown in Figure 2.24 are being read (note the first two rows of numbers are given to identify the columns) using the GET EDIT statement of Figure 2.25. Assume the computer is currently at column 1 of the first (top) card. The computer executes the statement as follows. It first checks for any OPTIONs that might be specified. No OPTIONs are given so it looks in the list of variables for some variable to assign a value to. It finds the variable A. It then looks in the format area for some way to input a value for A. The first specification that is found says to skip over three columns of the card from the current position. This places the card reader at column 4 on the first card. A value for A has not been read so the computer examines the next format specification. This specification states to read the next three columns as a fixed decimal number with no fractional digits. The number 3.4 is read from these columns. Because the

format specification said to read the number with no fractional digits, the number is converted to 3 and assigned to the variable A. A check is now made to see if more values should be read and it finds that a value should now be read for B. The computer continues on with the format specifications where it left off and finds CCL(13). This tells the card reader to go to column 13 on the current card. The next format specification states to read a

```
GET EDIT(A,B,C)(X(3),F(3),COL(13),F(10,3),SKIP,X(2),
                F(1),X(1),SKIP);
```

FIGURE 2.25 - SAMPLE GET EDIT STATEMENT

fixed decimal number 10 columns wide. Three of digits read should be fractional. The number 478.369234 is read. Because there is a decimal point in the number, it is used. This results in a value of 478.369 being assigned to B. If the decimal point had not been present in the columns read the computer would have placed a decimal point in the number so that the last three digits would be fractional.

A check is again made to see if more values should be input for variables in the list and it is discovered that a value for C should be read. The computer continues in the format specifications where it left off and finds SKIP. This causes the remaining values on the current card to be ignored. The card reader skips over them and procedes to the start of the next card. An X(2) specification is then found which causes columns 1 and 2 of the current card to be ignored, then an F(1) specification is found. This causes the value in the third column to be read as a fixed decimal number. Looking at that data card, notice that column 3 is blank. This is interpreted as a zero so C is given a value of zero. The computer again checks to see if variables still remain in the list and discovers none remaining. Because no more values should be read, the computer now procedes to execute the next statement of the program. Note that the remaining format specifications are ignored! This is because no more variables are to be assigned values.

Suppose fewer format specifications are given than variables

to be read, what happens? Assume that the GET statement now to be executed is the statement shown in Figure 2.26. After executing

```
GET SKIP EDIT(D,E)(F(3),X(4));
```

FIGURE 2.26 - GET EDIT STATEMENT WITH TOO FEW FORMAT SPECIFICATIONS

the SKIP causing the card reader to go to the third data card, the computer reads a value for the variable D. The format specifications state that that value should be read from columns 1 through 3 on the card giving a value of 937 to the variable. In attempting to read a value for E the computer skips over the next four columns of the data card and then runs out of specifications. Because a value must be read, the computer goes back to the start of the format specification list and starts through the list a second time. This causes a value of 149 to be assigned to E and leaves the card reader on the 10th column of the third data card.

Figure 2.27 shows several possible ways of writing the same GET EDIT statement. The first statement shows all of the format

```
GET EDIT(A,B,C)(X(2),F(3),X(2),F(3),X(2),F(3));

GET EDIT(A,B,C)(X(2),F(3));

GET EDIT(A,B,C)((3)(X(2),F(3)));
```

FIGURE 2.27 - VARIOUS WAYS OF WRITING THE SAME GET EDIT STATEMENT

specifications enumerated. When the list is highly repetitive, variations such as the second and third statements can be used. In the second statement the format specifications that are to be repeated are listed once. The computer will repetitively execute

the specifications until all variables have been assigned values. The third statement shows how a repetition factor can be used. The number 3 in parentheses states that the following group of format specifications (also given in parentheses) should be repeated three times. The use of the repetition factor only applies to the format specifications and can be used extensively as shown by Figure 2.28.

```
GET SKIP EDIT(A,B,C,D,E,F)((2)( (2)(F(4)),SKIP ),
    (2)(X(4),F(7,3)));
```

FIGURE 2.28 - EXTENSIVE USE OF THE REPETITION FACTOR IN A GET EDIT STATEMENT

The third type of GET statement is the GET DATA statement. The list of variables is optional in this statement, and no format specifications are given. The data cards for the GET DATA statement consist of a series of assignment statements separated by commas. The last assignment statement is followed by a semicolon. Figure 2.29 shows how such a data card might look. Note that only constants can occur to the right of the equal signs on the data cards. If no list of variables is given, the

```
A = 23, B = 17, C = 97, D = 57;
```

FIGURE 2.29 - SAMPLE DATA CARD FOR A GET DATA STATEMENT

computer will read to the first semicolon in the data cards, assigning values to the variables as specified. Figure 2.30 shows two GET DATA statements which will read in the data from the sample data card shown in Figure 2.29.

The second GET DATA card in Figure 2.30 specifies what variables can be given values when it is executed. The variables do not have to appear on the data card in that specified order nor must every variable specified be given a value. If 'C=97' was

removed from the sample data card of Figure 2.29, the statement

```
GET DATA;

GET DATA(A,B,C,D);
```

FIGURE 2.30 - SAMPLE GET DATA STATEMENTS

would assign the specified values to A, B, and D, leaving C with its current value. The data card, however, cannot contain variables that are not specified in the list. If the data card contained 'P=24' in place of 'C=97' an error would occur when the second GET DATA statement is executed, because P did not appear with A, B, C, and D in the list.

2.8.2 PUT STATEMENT

The PUT statement is used to output or print information from the program. The general forms of the PUT statement are given in Figure 2.31. The OPTIONs that are available to be used with the PUT statement are as follows:

FILE(FILE_NAME) - this allows the programmer to specify where the output should go. Normally SYSPRINT is used which causes the output to be printed on the line printer. If this option is not specified, SYSPRINT is automatically assumed.

SKIP {(EXP)} - this works identically to SKIP on input except printer lines are skipped rather than cards. A SKIP(0), however, will cause the printer to start printing at the beginning of the current line. This is used for overprinting of characters.

PAGE - this causes the computer to stop printing on the current output page and to go to the top of the next

page before continuing to print information.

```
FORM 1 - PUT [OPTIONS] 'POSITION' ;

FORM 2 - PUT [OPTIONS] LIST(EXPRESSION
         [, EXPRESSION] );

FORM 3 - PUT [OPTIONS] EDIT(EXPRESSION
         [, EXPRESSION] )( FORMAT );

FORM 4 - PUT [OPTIONS] DATA (VARIABLE
         [, VARIABLE] );
```

FIGURE 2.31 - GENERAL FORMS OF THE PUT STATEMENT

LINE(EXP) - this allows the programmer to specify on exactly which line on the page to print information. The expression, EXP, is evaluated and the computer then goes to that line on the page to print. Note, if the computer is currently writing on line 6 and LINE(5) is specified the computer will go to line 5 of the next page. This is because the line printer cannot back up.

Form 1 of the PUT statement, given in Figure 2.31, has only a POSITION specified along with some OPTIONs. A POSITION is any one of the last three OPTIONs available which are listed above. Any or all of the others may be used as OPTIONs.

The second form of the PUT statement has a series of expressions specified to be output in a list form, a maximum of six values per line. The computer divides the output line into six areas that are approximately equal in size. In executing the PUT LIST statement the computer evaluates the first expression and then outputs its value in the first available area beyond the printer's current position. Each expression is evaluated in turn and output in the same way, by going to the next available printing area. If the value of a given expression is too large for a single area it is continued into the second area. When the next expression is evaluated, its value is placed in the next (third) area.

The EDIT form of the PUT statement works in the same way as the GET EDIT. The expressions are evaluated then output according to the format specification given. Two additional positional format specifications are allowed:

PAGE - causes the computer to go to the top of a new page.

LINE(EXP) - causes the computer to go to the line number specified. (See above for more detail.)

The COLUMN and A specifications change to the following:

COL(EXP) - causes the computer to go to column number specified. If that column number has been passed on the current line, it goes to that column number on the next line.

A[(EXP)] - if the expression is present, it will be evaluated as an integer and that many characters will be output. If there are more characters given to be output than spaces available, the extra characters will be removed from the right of the character string. If there are less characters than places, the extra places will be filled by blanks on the right. If only A is specified with no expression, all of the characters in the string will be output.

The PUT DATA statement causes the values of the variables to be printed out as assignment statements, separated by commas with a semicolon on the end.

Note that the OPTION (SKIP, LINE, PAGE) will always be done first in a PUT or GET statement. If more than one OPTION is specified, the order of execution is: PAGE, LINE, SKIP.

2.9 GO TO STATEMENT

The GO TO statement is used to transfer from one point to another point in a program. Figure 2.32 shows the general form of a GO TO statement. LABEL is a label identifier used within the program.

Figure 2.33 shows how a GO TO statement might be used in a

```
GO TO LABEL;
```

FIGURE 2.32 - GENERAL FORM OF A GO TO STATEMENT

```
      :
      :
HERE:X=A*B;
     C=A*A;
      :
      :
GO TO HERE;
HALT:A=0;
      :
      :
```

FIGURE 2.33 - EXAMPLE OF THE USE OF A GO TO STATEMENT

program. The statements in the program are executed in their normal numeric order until the GO TO statement is encountered. Rather than continuing execution with the statement labeled HALT the computer will transfer back to the statement labeled HERE and will continue execution at that point.

The use of GO TO statements in programs should be limited as much as possible. The excessive use of GC TO statements causes the program to be very difficult to read and understand. If an alternative is available to using a GO TO statement, use it rather than the GO TO. Do not, however, destroy a well written, easy to understand program just to eliminate one GO TO statement. As a rule of thumb, if 10 percent of a program is GO TO statements, the program should be rewritten.

2.10 IF STATEMENT

The IF statement allows the programmer to test conditions during the execution of the program so the program can determine, by the results of testing the condition, what action should be performed next. The general form of an IF statement is given in Figure 2.34. The CONDITION is some logical expression which

```
IF CONDITION THEN S1; { ELSE S2; }
```

FIGURE 2.34 - GENERAL FORM OF THE IF STATEMENT

results in a true or false value when evaluated. S1 and S2 are any executable statements in the language. If the evaluation of the condition results in a value of true, statement S1 is executed. If the result is false and an ELSE clause is given, statement S2 is executed. After executing either the ELSE or THEN clause of the IF statement, the computer continues execution with the statement following the IF statement. If no ELSE clause exists for the statement and the condition is false, the next statement is executed as though the ELSE clause was a null statement. The only case when the next statement is not executed is when the ELSE or THEN clause that is executed contains a GO TO statement. The computer in this case goes to the specified label and continues execution at that point as though it was the statement following the IF statement.

The conditional operations that can be used in the condition of an IF statement are given in Figure 2.35. All arithmetic operators have a higher precedence than these conditional operators so these conditional operators will be executed last.

Suppose that the absolute value of a number was desired.

$$A = | X | \qquad (2.12)$$

This could quite easily be expressed with the following IF

=	EQUAL TO
>	GREATER THAN
<	LESS THAN
¬=	NOT EQUAL
¬>	NOT GREATER THAN
¬<	NOT LESS THAN
>=	GREATER THAN OR EQUAL TO
<=	LESS THAN OR EQUAL TO

FIGURE 2.35 - CONDITIONAL OPERATORS

statement:

```
IF X>0 THEN A=X;                                    (2.13)
       ELSE A=-X;
```

Suppose, however, that this should only be done if some variable P has a value of 3. This could be written as

```
IF P=3 THEN IF X>0 THEN A=X;                        (2.14)
                   ELSE A=-X;
```

Immediately the question is asked - which IF statement does the ELSE belong to? ELSE clauses are always matched to the closest IF statement that does not have an ELSE clause. For example -

```
IF P=3 THEN IF X=0 THEN A=X;
                   ELSE A=-X;                       (2.15)
       ELSE IF H>0 THEN A=H+X;
                   ELSE A=-H+X;
```

Notice the use of indentation to show graphically how the ELSE clauses are matched. This IF statement can be diagrammed as:

```
                          ┌- THEN A=X
          ┌- THEN IF X>0 -┤
          |               └- ELSE A=-X
          |
IF P=3 -┤                                                    (2.16)
          |
          |               ┌- THEN A=H+X
          └- ELSE IF H>0 -┤
                          └- ELSE A=-H+X
```

Suppose it was desired to place three numbers A, B, and C in numeric order so that A > B > C. This could be written as one large compound or nested IF statement as follows:

```
                                          ┌- THEN A>C>B
                          ┌- THEN IF A>C -┤
                          |               |
                          |               └- ELSE C>A>B
          ┌- THEN IF B<C -┤
          |               |
          |               └- ELSE IN ORDER
IF A>B -┤                                                    (2.17)
          |               ┌- THEN B>A>C
          |               |
          └- ELSE IF A>C -┤
                          |               ┌- THEN B>C>A
                          |               |
                          └- ELSE IF B>C -┤
                                          |
                                          └- ELSE C>B>A
```

This diagram becomes the following IF statement -

```
IF A>B THEN IF B<C THEN IF A>C THEN DO;
                                    T=C;
                                    C=B;
                                    B=T;
                                    END;
                               ELSE DO;
                                    T=C;
                                    C=B;
                                    B=A;
                                    A=T;
                                    END;
                    ELSE;
          ELSE IF A>C THEN DO;                                    (2.18)
                           T=B;
                           B=A;
                           A=T;
                           END;
                    ELSE IF B>C THEN DO;
                                    T=B;
                                    B=C;
                                    C=A;
                                    A=T;
                                    END;
                               ELSE DO;
                                    T=C;
                                    C=A;
                                    A=T;
                                    END;
```

Notice the null ELSE clause in the middle of the statement. It is very tempting to remove that ELSE clause because it does nothing, but if it is removed the meaning of the statement is dramatically changed. This is due to the rule that ELSE clauses are matched to the nearest IF statement that does not already have an ELSE clause. If the null ELSE clause is removed, the ELSE that follows it will be put in its place, matching up with the 'IF B<C' IF statement. Never remove null ELSE statements for this reason.

Compound conditions can also be used in IF statements. The logical operators are -

```
¬      NOT
&      AND
|      OR
```

NOT has the highest precedence and OR the lowest. A sample compound conditional IF statement is given below.

```
IF A=B&(C=D | P>3) THEN C=1;
                   ELSE C=3;                    (2.19)
```

Parentheses can be used as shown to change the precedence as is done with the arithmetic operators in assignment statements.

2.11 NULL STATEMENT

The NULL statement consists simply of a semicolon. Its execution results in nothing being done. NULL statements are normally used for placing labels in programs

```
    :
    :
LABEL:;                                         (2.20)
    :
    :
```

and for insuring that nested IF statements are executed properly.

2.12 PROCEDURE STATEMENT

The first executable statement in a program must be a PROCEDURE statement. The general form of a PROCEDURE statement is given in Figure 2.36. NAME is a label consisting of up to seven alphanumeric characters (letters or numbers) which starts with a

letter. This is the name of the program and is usually choosen to

```
NAME:PROCEDURE OPTIONS(MAIN);
```

FIGURE 2.36 - GENERAL FORM OF A PROCEDURE STATEMENT

reflect what the program actually does. NAME is followed by a colon and the word PROCEDURE or PROC, its abbreviation. OPTIONS(MAIN) states that what follows is the main procedure or main program, the place where the computer should start executing instructions. A sample procedure statement might be -

```
SORT:PROCEDURE OPTIONS(MAIN);                    (2.21)
```

This might be the PROCEDURE statement for a program that sorts numbers into numeric order.

For other forms of the PROCEDURE statement see Chapter 6 on procedures.

2.13 RETURN STATEMENT

The RETURN statement is used to specify the value being returned from a function procedure or to end an invoked procedure before its logical end. RETURN statements used in function procedures specify in parentheses the value that should be returned. The following RETURN statement specifies that the value of the variable A should be returned -

```
RETURN(A);                                       (2.22)
```

Expressions and constants can also be used, for example -

```
RETURN(3.7);                                     (2.23)
```

or

```
RETURN(27*F(I));                                    (2.24)
```

A called procedure has no value that should be returned, so none is specified -

```
RETURN;                                             (2.25)
```

The RETURN statement can be used anywhere in an invoked procedure. It can be thought of as being a STOP statement for that procedure, causing its execution to be halted and execution to resume in the main program. If a RETURN statement does not appear in a referenced procedure, the computer will return automatically when the END statement for the procedure is encountered. Whenever a RETURN statement is used or not, the computer returns to the statement that caused the procedure to be invoked and continues execution at that point.

2.14 STOP STATEMENT

The STOP statement is used to stop the execution of the program at some point other than its logical end (the END statement corresponding to the main procedure statement). It may be used anywhere in the program and will result in the program's execution being halted when it is encountered. Normally, the STOP statement is used in an IF statement to end the program is some condition is true. For example

```
TEST:PROCEDURE OPTIONS(MAIN);
   :
   :
 IF X=0 THEN STOP;                                  (2.26)
   :
   :
 END TEST;
```

This is much clearer and easier to read than

```
TEST:PROCEDURE OPTIONS(MAIN);
   :
   :
 IF X=0 THEN GO TO LAST;                    (2.27)
   :
   :
 LAST:END TEST;
```

CHAPTER 3 - ARRAYS AND STRUCTURES

Because programs are desired to handle large amounts of data that is very similar, better ways of storing the information must be used than simple variables. An obvious example of this is the list of numeric grades received by students in a course. It would be very tedious to write a program using simple variables to print out the grades in order from highest to lowest. Many variables would be needed and the program would only work for that number of variables. If another student added the course, the program would have to be entirely rewritten which would be quite wasteful. The use of arrays and structures can greatly simplify this problem.

3.1 ARRAYS

Arrays can be thought of as a large number of variables that share the same name. The DECLARE statement of Figure 3.1 defines an array, A, which has 15 elements or members. To reference or use any single element of array A, the name of the array is used with a number specifying which element to use from the array. For example

```
A(5)                                    (3.1)
```

references element number 5 in the array, while

```
A(I)                                    (3.2)
```

```
DECLARE A(15) FIXED DECIMAL INITIAL((5)(0),
              (5)(1,2));
```

FIGURE 3.1 - ARRAY DECLARATION STATEMENT

references the Ith element.

Note that initial values are given to array A in Figure 3.1. The first 5 elements are assigned a value of zero while the last 10 elements are assigned alternating values of one and two. These initial values are assigned to the elements of the array before the execution of any statements in the program. The initialization is done only once.

In declaring arrays the bounds or limits of the array's dimensions are left to the programmer to decide. A general form of the DECLARE statement is shown in Figure 3.2. LBI is the lower

```
DCL NAME (LB1:UB1, LB2:UB2, ..., LBN:UBN);
```

FIGURE 3.2 - GENERAL FORM OF A DECLARATION OF AN ARRAY

bound of the Ith subscript (dimension) and UBI is the upper bound of the Ith subscript. If a lower bound is not given it is assumed to be 1. There is no limit to the number of subscripts that can be used but there are very few applications that require more than three or four subscripts.

Suppose a program was desired to compare various automobile manufacturers. A possible array used in the program might hold the number of cars produced by each of 5 manufacturers during the years 1960 through 1970. An array to hold this data might be declared as

```
DCL CARS(5,11) FIXED DECIMAL;                    (3.3)
```

where the first subscript refers to the manufacturer and the second to the year. Declaring the array in this way presents a problem to the programmer. He must remember that CARS(1,5) refers to the cars produced by manufacturer number one in the year 1964. This can lead to many problems in understanding the program. Instead the DECLARE statement for this array could be

```
DCL CARS(5,1960:1970) FIXED DECIMAL;             (3.4)
```

Now the programmer can refer to the cars produced by the first manufacturer in the year 1964 as CARS(1,1964) eliminating confusion in writing and understanding the program.

Once arrays have been declared, values can be placed in the elements of the arrays. The subscripts used to refer to specific elements of the arrays can be expressions. Because expressions can be used as subscripts it is quite easy to write programs to manipulate the values in arrays. If a total for the number of cars produced each year in the above example was desired, it could easily be computed as shown in Figure 3.3. Arrays can also be used without subscripts when an operation is to be performed on all of the elements of the array. If, for example, all elements in array A, declared in Figure 3.1 above, were to be incremented by 11, it could be expressed as

```
A = A + 11;                                      (3.5)
```

rather than

```
DO I=1 TO 15;
  A(I) = A(I) + 11;                              (3.6)
  END;
```

While this can save some programming effort it does tend to lead to confusion when 'reading' a program and many problems can only be done using a DO loop as shown in Figure 3.3 because of restrictions in the PL/I language. Suppose, for example, is was

desired to multiply array A by B and place the results in C. If

```
/* LOOP TO GO THRU THE YEARS          */
DO I=1960 TO 1970;
  /* INITIALIZE A SUM TO ZERO         */
  SUMM=0;
  /* LOOP FOR THE MANUFACTURERS       */
  DO J=1 TO 5;
    /* ADD NUMBER OF CARS INTO TOTAL  */
    SUMM=SUMM + CARS(J,I);
    END;
  /* PRINT YEAR AND NUMBER OF CARS    */
  PUT SKIP LIST('THE YEAR', I, ' SAW', SUMM,
               ' CARS PRODUCED.');
  END;
```

FIGURE 3.3 - USE OF DO LOOPS TO SUM ELEMENTS IN AN ARRAY

the arrays are declared to be

```
DCL (A(5), B(5), C(0:5)) FLOAT DEC;                    (3.7)
```

then the multiplication must be done as follows:

```
DO I=1 TO 5;
  C(I-1)= A(I)*B(I);                                   (3.8)
  END;
```

This is due to the fact that the bounds on array C are not exactly the same as the bounds on A and B, causing

```
C=A*B;                                                 (3.9)
```

to be unclear to the computer. If in doubt as to what will happen, always use a loop, rather than the implied single statement.

3.2 STRUCTURES

One disadvantage of arrays is that they can hold only one type of information. An array declared FIXED BINARY can only hold fixed binary numbers - not character strings, and vice versa. Many programming applications require that data of different types be associated with each other. For this reason, structures were created.

Suppose a program was desired to sort a list of grades that students made in a course into numeric order. In getting grades in numeric order, it would also be advantageous to have a list of which student made what grade. One could easily set up two arrays, one to hold the character strings of student names and the other to hold the numeric grades. As more and more information on the students is desired to be kept (like home address, guardian's name, etc.), this becomes increasingly more difficult to manage. It would be quite advantageous to declare something called STUDENT that would contain all of this information. For each student would be a place for a name, grade, etc... . Such a structure is shown in Figure 3.4. This structure

```
DCL 1 STUDENT(20),
      2 NAME CHARACTER(15),
      2 GRADE FIXED DECIMAL,
      2 HOME_ADDR CHARACTER(30);
```

FIGURE 3.4 - SAMPLE STRUCTURE CONTAINING STUDENT INFORMATION

contains two levels. The uppermost level (level 1) defines the name of the structure (STUDENT). Each of the sublevel identifiers in this structure contain information like a given student's name. A dimension is given to the top level of the structure so

that 20 students structures will be created, each containing the student's name, grade, and home address. The structure numbers, from one level to the next, do not need to increase in exact numeric order. The uppermost level in the above structure could

```
DCL 5 STUDENT(20),
      10 HOME_ADDR CHAR(30),
      10 COURSE(5),
        15 C_NAME CHAR(10),
        15 GRADE FIXED DEC,
      10 S_NAME CHAR(15);
```

FIGURE 3.5 - STRUCTURE CONTAINING STUDENT AND COURSE INFORMATION

just as easily been numbered 5 instead of 1. Each sublevel, however, must have larger number than the level above it, so NAME, etc. would have to have structure numbers that were greater than 5. Whenever the structure number increases numerically a new sublevel is being defined. The previous structure element has to consist of just a name for that substructure. For example, suppose each student has 5 courses and receives a grade in each. The resulting structure might be as shown in Figure 3.5. This structure now has three levels. The uppermost level defines the entire structure, names it STUDENT, and states that 20 such structures exist. On the second level three identifiers are defined. HOME_ADDR is a character string of 30 characters, S_NAME is a character string of 15 characters, and COURSE is a substructure. Contained within the COURSE structure is C_NAME, a character string of length 10, and GRADE, a fixed decimal number. Note that each STUDENT structure contains five COURSE substructures, each for a different course that the student is taking.

In addition to the STUDENT structure, the program might contain a structure for courses as shown in Figure 3.6. Note that there now exists two variables called C_NAME. This situation arises often in structures, so methods must be used to distinguish between two course name. To do this qualifiers are

added to the variable name to specify which variable it is. Any

```
DCL 1 COURSES(20),
      2 C_NAME CHAR(30),
      2 CALL_NUM FIXED DECIMAL(6),
      2 INST_NAME CHAR(30),
      2 HOURS FIXED DECIMAL(2);
```

FIGURE 3.6 - COURSE STRUCTURE

of the following qualified names could be used to reference the student's course's name -

```
STUDENT(I).COURSE(J).C_NAME
STUDENT(I).COURSE.C_NAME(J)
STUDENT.COURSE(I,J).C_NAME                    (3.10)
STUDENT.COURSE(I).C_NAME(J)
STUDENT.COURSE.C_NAME(I,J)
```

The placement of the subscripts is not significant, so long as their order remains the same and the subscripts do not occur in the qualified name before their natural place as defined by the structure. Had the course's name been declared in the COURSE structure as

```
2 CRS_NAME CHAR(30),                          (3.11)
```

the following additional qualified names could be used to reference the student's course's name -

```
COURSE(I,J).C_NAME
COURSE(I).C_NAME(J)                           (3.12)
COURSE.C_NAME(I,J)
C_NAME(I,J)
```

As a rule, always qualify the variable enough so that it cannot be confused with any other.

As with arrays, operations can be performed on whole structures.

```
STUDENT(I) = STUDENT(J);                    (3.13)
```

The above statement causes all information in the student structure associated with student J to be placed in the structure for student I. Care must be taken as with arrays that the structures are totally the same and that meaningless statements, e.g.

```
STUDENT(I) = STUDENT(I) + 15;               (3.14)
```

are not used.

CHAPTER 4 - SEARCHING AND SORTING

In this chapter, several searching and sorting techniques that are useful when working with arrays and structures are examined. The algorithms discussed in this chapter are a few of the basic algorithms that should be known by all computer science students. Many additional searching and sorting techniques exist. For additional information consult THE ART OF COMPUTER PROGRAMMING, VOL. 3 by D. E. Knuth.

4.1 LINEAR SEARCH

Suppose that the student grade structure (discussed in the last chapter) was stored within the machine. A program that might be of use for a school administration would be one to locate a given student's records. This searching might use the following basic algorithm -

1) Set I=1 (pointer to the first element in the structure).
2) Is I greater than the number of elements in the structure? If so, the item being searched for does not exist. We are done.
3) Is element(I) the item we are looking for? If so, we are done.
4) Set I=I+1 (set pointer so it points at next element in the structure).
5) Go to step 2.

```
/* HUNT_NAME - VARIABLE THAT HOLDS NAME OF STUDENT */
/*             BEING LOOKED FOR                    */
/* J - VARIAELE THAT WILL HOLD POSITION OF THE     */
/*             STUDENT IN THE STUDENT STRUCTURE.   */
/*             IF STUDENT IS NOT FOUND, J WILL     */
/*             EQUAL ZERO.                         */
J=0;

/* DO A LOOP FROM THE FIRST STUDENT THRU THE LAST  */
LOOP:DO I=1 TO N BY 1;

/* IS THIS THE STUDENT WE ARE LOOKING FOR?         */
   IF STUDENT(I).S_NAME=HUNT_NAME THEN DO;

/* SET VALUE OF J AND LEAVE LOOP                   */
      J=I;
      GO TO OUT;
      END;
   END LOOP;
OUT:;
```

FIGURE 4.1 - LINEAR SEARCH PROGRAM

This algorithm is known as a linear search because it starts at the first element of the structure and proceeds element by element through the structure until it finds the item being looked for or discovers that the item is not in the structure. Sometimes the item that is being looked for will be found in the first element of the structure, other times the item will be found in the last element of the structure, but on an average, this algorithm will be forced to look at one-half of the elements of the structure before finding the location of the item being searched for. If the item is not in the structure, the search will always go through every item of the structure. Translating the algorithm into statements results in the program shown in Figure 4.1.

Searching in this manner can prove to be very slow. If the structure contains 40,000 students, an average of 20,000 comparisons would have to be made to find one student's records, and if the record was not there it would take 40,000 comparisons. If the computer can make 1000 comparisons a second, looking for one student's records would take an average of 20 seconds.

Suppose, however, that the students were listed in alphabetical order. Would this help to decrease the search time? Using the linear search, some times the item will again be the first element in the list, and other times it will be the last element. On an average it will still take N/2 comparisons (where N is the number of students in the list). When an item is not in the list, the number of comparisons will only be (N/2)+1 rather than N. This is because of the following: as an item is looked for in the structure, the elements from the structure that it is compared to are always less than the item. This holds until the item is found in the structure. If the item was not in the structure, it will then start being compared against elements from the structure that are greater than it is. When this first element is found that is greater than it is, there is no point in continuing because that item cannot occur later in the structure. For example, suppose the position of the number 23 was desired in the following array.

```
A(1) = 10
A(2) = 12
A(3) = 20
A(4) = 21                                   (4.1)
A(5) = 23
A(6) = 47
A(7) = 59
A(8) = 71
```

It would take 5 comparisons to locate it in the array, because it is located in the 5th element of the array. Suppose, however, the location of 25 was desired. The comparisons would continue in the same way as with the number 23, except the 6th comparison would show that the array elements have now become greater than the item being looked for. The search need not continue beyond this point because then 25 cannot occur beyond element 6 of the array. The savings of having the elements sorted is not significant when the linear search is used; however, there is a method that is much faster than the linear search because it makes use of the fact that the numbers are sorted.

4.2 THE BINARY SEARCH

Rather than looking linearly at the elements of a sorted structure, why not look at the middle element. By looking at the middle element of the structure, if the item is not that element, one half of the elements in the structure can then be ignored in the next comparison. For example, suppose the position of 21 was desired in the previous array, A. The middle element is element number 4 or 21 and the item is found in only 1 comparison rather than 4. Suppose that 23 had been desired instead. Element number 4 or 21 would first be examined. Because element 4 is less than the item being searched for, all elements from 4 down can be ignored in the next comparison because 23 cannot occur down there. The midpoint of the remaining items is then found. This is found to be element number 6 or 47. Because 47 is larger than the item being searched for, the comparisons can continue ignoring all elements from number 6 on up. The only element that remains is number 5 or 23, so the comparison is made and the item is found in only 3 comparisons instead of the 5 when using the linear search.

While this savings in comparisons made may not seem significant in this example, imagine what it would be with 40,000 elements in the structure. The first comparison would eliminate 20,001 from consideration. The next comparison would eliminate 10,000 more. In 10 comparisons only 40 elements of the original 40,000 would still be in consideration. In a maximum of 16 comparisons, the item will have been found. This results in an average savings of at least 19,984 comparisons. The maximum number of comparisons made using the binary search is LOG2(N+1) on a list containing N elements. The steps to the binary search algorithm are:

1) Set FIRST=1.
2) Set LAST=N.
3) If FIRST is greater than LAST then stop - the item is not in the list.
4) Set MID= (FIRST+LAST)/2.
5) If A(MID)=ITEM then stop the item is found.
6) If A(MID)<ITEM then throw out the bottom half (set FIRST=MID+1) otherwise throw out the top half (set LAST=MID-1).

7) Go to 3.

Writing this algorithm to hunt for a given student's records results in the program shown in Figure 4.2.

```
/* INITIALIZE FIRST, LAST AND J                          */
/* J WILL POINT TO THE NAME IN THE STRUCTURE OR WILL */
/* EQUAL ZERO IF IT IS NOT FOUND                         */
FIRST=1;
LAST=N;
J=0;
/* SET UP A LOOP TO GO AS LONG AS FIRST IS LESS THAN */
/* OR EQUAL TO LAST                                      */
DO WHILE(FIRST <= LAST);
  /* COMPUTE THE MIDPOINT AND CHECK TO SEE IF IT IS  */
  /* THE PROPER ITEM                                     */
  MID=(LAST+FIRST)/2;
  IF STUDENT(MID).S_NAME=HUNT_NAME THEN DO;
    J=MID;
    GO TO OUT;
    END;
  /* IF NOT, DETERMINE WHICH PART OF THE STRUCTURE TO */
  /* THROW OUT                                           */
  IF STUDENT(MID).S_NAME>HUNT_NAME THEN LAST=MID-1;
                                   ELSE FIRST=MID+1;
  END;
OUT:;
```

FIGURE 4.2 - BINARY SEARCH PROGRAM

4.3 JUMP DOWN (SIMPLE) SORT

The problem that now remains is how to get the elements of an array or structure in order so that the binary search can be

used. The most obvious way of doing this is to locate the largest element of the array and place it into the last position. This

```
/* LOOK AT ARRAY, ONE ELEMENT LESS EACH TIME          */
DO I=N TO 2 BY -1;
  /* SET A(1) TO BE THE LARGEST ELEMENT AND REMEMBER */
  /* WHERE IT IS                                     */
  MAX=A(1);
  J=1;
  /* NEW LOOK TO SEE IF IT IS THE LARGEST            */
  DO K=2 TO I;
    IF A(K)>MAX THEN DO;
      /* IT IS NOT THE MAXIMUM SO MAKE THIS ELEMENT  */
      /* THE MAXIMUM                                 */
      MAX=A(K);
      J=K;
      END;
  END;
/* SWAP MAXIMUM AND LAST ELEMENT OF THE CURRENT      */
/* SUBSECTION OF THE ARRAY                           */
TEMP=A(I);
A(I)=A(J);
A(J)=TEMP;
END;
```

FIGURE 4.3 - JUMP DOWN (SIMPLE) SORT

last position is then ignored and the process is repeated until only one item remains to be sorted and the algorithm stops. The basic steps are then:

1) Go through the array, first with all N elements, then with N-1 until only 1 element remains.
2) Go through the part of the array being examined looking for the largest element.
3) Swap the largest element and the last element of the current part of the array.
4) Repeat the steps.

This algorithm translates into the program given in Figure 4.3. Using the program of Figure 4.3, the sorting process progresses as follows on the array shown below.

```
A(1)=10
A(2)= 1
A(3)= 3
A(4)= 7
A(5)= 8
A(6)= 9
```
(4.2)

The first time through the outer loop, the largest element is found to be element 1. This element is then swapped with the last element giving -

```
A(1)= 9
A(2)= 1
A(2)= 3
A(4)= 7
A(5)= 8
A(6)=10
```
(4.3)

The program now looks only at the top 5 elements of the array and repeats the steps (the line is used to show the limits of elements being used to find the largest) -

```
A(1)= 8
A(2)= 1
A(3)= 3
A(4)= 7
A(5)= 9
-------
A(6)=10
```
(4.4)

This process continues to be repeated, resulting in the following values (the rightmost is the final answers) -

```
A(1)= 7     A(1)= 3     A(1)= 1
A(2)= 1     A(2)= 1     A(2)= 3
A(3)= 3     A(3)= 7     -------
A(4)= 8     -------     A(3)= 7
-------     A(4)= 8     A(4)= 8
A(5)= 9     A(5)= 9     A(5)= 9
A(6)=10     A(6)=10     A(6)=10
```
(4.5)

4.4 BUBBLE SORT

Examining the original array again it is noticed that the array is completely in order except the first element.

```
A(1) =10
A(2) = 1
A(3) = 3
A(4) = 7                                                    (4.6)
A(5) = 8
A(6) = 9
```

If that element was to be slowly moved down to its proper place, as follows from left to right, much less work is done -

```
A(1) =10   = 1   = 1   = 1   = 1   = 1
A(2) = 1   =10   = 3   = 3   = 3   = 3
A(3) = 3   = 3   =10   = 7   = 7   = 7
A(4) = 7   = 7   = 7   =10   = 8   = 8                      (4.7)
A(5) = 8   = 8   = 8   = 8   =10   = 9
A(6) = 9   = 9   = 9   = 9   = 9   =10
```

This algorithm is known as a bubble sort because the elements are slowly bubbled to their proper place. The algorithm steps are -

1) Go through all elements of the array looking at all elements the first time, then one less each successive time until only one remains.
2) If no swaps were made the last time through the array then stop because the array is in order.
3) Go through the part cf the array currently being examined, swapping adjacent elements if they are out of order.

These steps translate into the statements shown in Figure 4.4. This method is much faster than the previous method in certain cases. These cases are when the array is almost in order. But this method is still quite inefficient when the array is completely out of order.

```
/* MAKE IT APPEAR THAT ITEMS HAVE BEEN SWAPPED          */
SORTED=0;
/* LOOP THROUGH ARRAY MAKING IT SMALLER EACH TIME       */
DO I=N TO 2 BY -1;
  /* CHECK TO SEE IF DONE                               */
  IF SORTED=1 THEN GO TO DONE;
  /* ASSUME THINGS ARE NOW SORTED                       */
  SORTED=1;
  /* GO THRU CURRENT PART OF ARRAY                      */
  DO J=1 TO I-1;
    IF A(J)>A(J+1) THEN DO;
      /* ELEMENTS OUT OF ORDER SO SWAP                  */
      TEMP=A(J+1);
      A(J+1)=A(J);
      A(J)=TEMP;
      /* SET SORTED SO IT STATES THAT A SWAP WAS MADE */
      SORTED=0;
      END;
    END;
  END;
DONE:;
```

FIGURE 4.4 - BUBBLE SORT

4.5 HEAP SORT

One of the fastest method for sorting arrays is known as the heap sort. To understand how this sort works, one must first know what a binary tree is. A binary tree is a way of representing data in the computer. These trees are drawn upside down from the normal concept of a tree with the root node being at the top (see Figure 4.5). Nodes are where the data is stored and branches connect the nodes together. In a binary tree each node has two

branches leading from it to its descendants. The tree has only

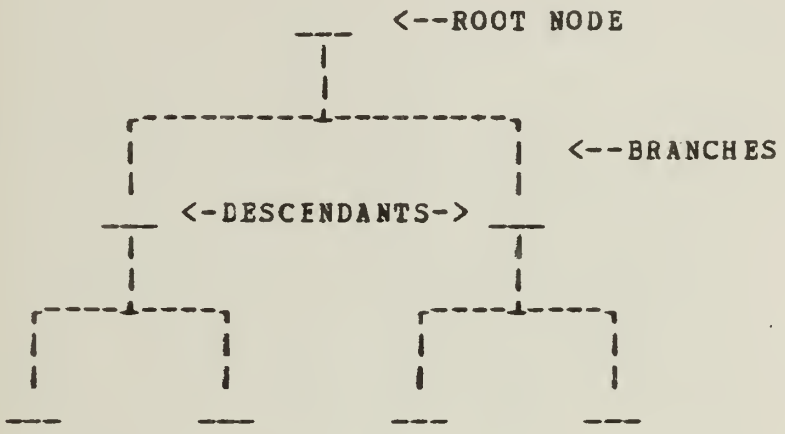


FIGURE 4.5 - GENERAL DIAGRAM OF A BINARY TREE

one root node which is said to be at level 1. The descendants of the root node are said to be at level 2. The descendants of these nodes are at level 3.

The descendants of any node are known as the right son and left son. The immediate ancestor of any node is known as its father. A convenient labeling for the nodes is to call a given father, node I. Its left son is node 2*I and the right son is node 2*I+1. This results in the labels shown in Figure 4.6. Note that by using this numbering scheme a binary tree can very easily be placed into an array. One restriction that must be applied is that only complete tree will be used in this sorting method. Complete trees are trees in which if node I exists then so does all nodes from 1 to I-1. In other words, there are no holes in the tree.

A heap is a binary tree in which each ancestor node is greater than or equal to its decendants. The complete binary tree shown in Figure 4.7 is a heap. The following calculations can be used to compute the father and sons of nodes -

```
Father of A(I) is A(I/2)
Left Son of A(I) is A(2*I)
Right Son of A(I) is A(2*I+1)
```

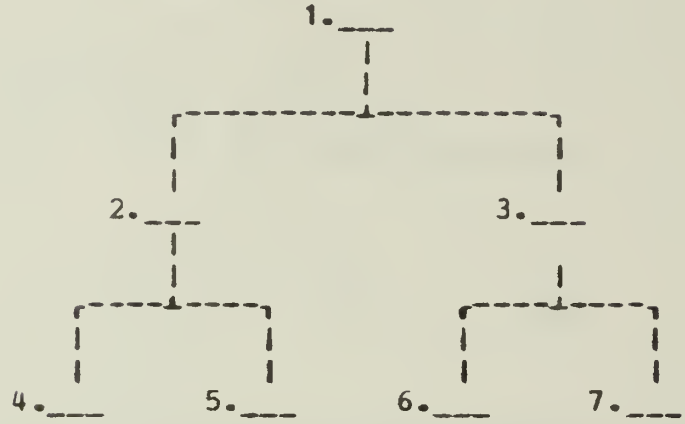


FIGURE 4.6 - GENERAL DIAGRAM OF A BINARY TREE WITH LABELED NODES

The steps to the heap sort algorithm are:

1) Make the array into a heap.
2) Swap the root node with the last node in the tree.
3) Eliminate the last node from the tree.
4) Make the tree into a heap.
5) Repeat steps 2 through 4 until only the root node remains.

First, a heap must be made. This can be done using the following steps -

1) Start with two nodes.
2) Make a heap.
3) Add the next node.
4) Repeat steps 2 and 3 until there are no nodes left.

A useful property of trees is that "if a tree is a heap except

for one node then exchanging the father of that node with the

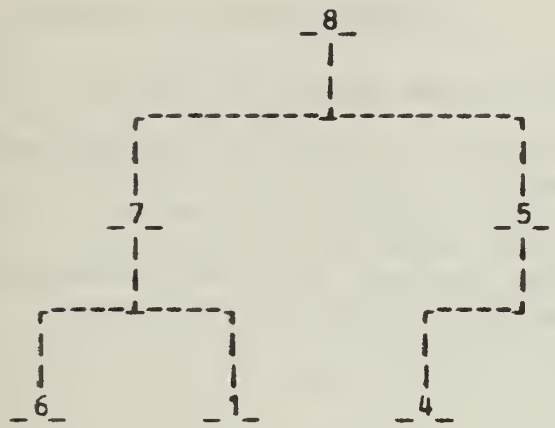


FIGURE 4.7 - SAMPLE COMPLETE BINARY TREE THAT IS A HEAP

node will cause the only offending relation to be between that father node and his ancestors". This results in the program of Figure 4.8. Now that the heap has been formed, it must be sorted. The following algorithm will sort the heap.

1) Swap the root node (A(1)) and the last node in the tree (A(I)).
2) Eliminate A(I) from the tree.
3) Make the tree into a heap by bubbling the element that was just placed in the the root node down as far as it will go.
4) Repeat steps 1 to 3 until there is only one node left in the tree.

This algorithm becomes the program segment shown in Figure 4.9. Note that in making the heap, the number of comparisons made is LOG2(N), and that in sorting the heap there is at most 2*LOG2(N) comparisons. This sort is then of order LOG2(N).

```
/* ADD NODES ONE AT A TIME STARTING WITH TWO     */
DO I=2 TO N;
  /* LET J POINT TO THE NODE JUST ADDED          */
  J=I;
  /* AS LONG AS J HAS A FATHER DO THE FOLLOWING */
  DO WHILE(J/2>=1);
    /* CHECK TO SEE IF TREE IS A HEAP            */
    IF A(J/2)>=A(J) THEN GO TO OUT;
    /* SWAP FATHER AND SON AND REPEAT PROCESS    */
    /* WITH FATHER                               */
    TEMP=A(J/2);
    A(J/2)=A(J);
    A(J)=TEMP;
    J=J/2;
    END;
  OUT:END;
```

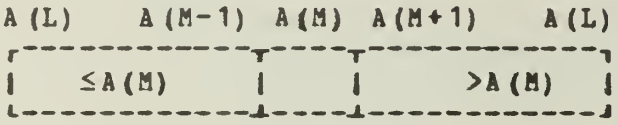
FIGURE 4.8 - PROGRAM SEGMENT TO FORM A HEAP

```
/* START WITH N ELEMENTS                                    */
I=N;
/* GO AS LONG AS THERE IS MORE THAN 1 ELEMENT               */
/* LEFT IN THE TREE                                         */
DO WHILE(I>1);
  /* SWAP THE ROOT NODE AND LAST NODE                       */
  TEMP=A(1);
  A(1)=A(I);
  A(I)=TEMP;
  /* ELIMINATE LAST NODE FROM THE TREE                      */
  I=I-1;
  /* MAKE A HEAP FROM THE TREE                              */
  J=1;
  K=2*J;
  /* KEEP GOING AS LONG AS J HAS A SON                      */
  DO WHILE(K<=I);
    /* FIND THE LARGEST SON                                 */
    IF K<I THEN IF A(K)<A(K+1) THEN K=K+1;
    /* CHECK TO SEE IF TOP NODE AND LARGEST                 */
    /* SON ARE IN ORDER                                     */
    IF A(J)>A(K) THEN GO TO EXIT_MAKE_HEAP;
    /* NOT IN ORDER SO SWAP ELEMENTS AND REPEAT             */
    TEMP=A(J);
    A(J)=A(K);
    A(J)=TEMP;
    J=K;
    K=2*J;
    END;
  EXIT_MAKE_HEAP:END;
```

FIGURE 4.9 - PROGRAM SEGMENT TO SORT THE HEAP

4.6 QUICKSORT

Another fast method for sorting arrays is known as the quicksort. A basic part of the algorithm is the process of partitioning the array into three segments as shown below:



Once the array has been partitioned, the two partitions, from A(L) to A(M-1) and A(M+1) to A(U), can each be partitioned. Eventually, the partitions will each contain only one or two elements of the array. If the partition contains only one array element nothing needs to be done. If it contains two array elements, a check is made to make certain they are in their proper order and if they are not they are swapped.

To start the partitioning process an A(M) value must be choosen. This value should be one that will, hopefully, evenly partition the other values in the array. Because there is no way of knowing what the values in the array are, one of the values must be choosen under the assumption that it is the middle value of all values in the array. The first value in the partition is choosen for lack of a better choice.

The partitioning algorithm is given in Figure 4.10. Basically, what happens is that two pointers into the array are used, M and N. All values from A(L+1) to A(N-1) are less than the value of A(L), the choosen middle value. All values from A(M+1) to A(U) are greater than A(L). The pointers are moved closer and closer together till they pass each other. The pointer N is moved until a value is found larger than A(L) then the pointer M is moved until a value is found less than A(L). These values are swapped and the process is repeated until M<N. This occurs when the array has been partitioned.

Suppose that the algorithm is applied to the following array -

```
/* SET VALUES OF N AND M                          */
M=U;
N=L+1;
/* LOOP TO EITHER INCREASE VALUE OF N OR          */
/* DECREASE VALUE OF M PARTITIONING THE ARRAY     */
DO WHILE(N<=M);

  IF A(L)>=A(N) THEN N=N+1;
    ELSE DO;
    /* DECREASE VALUE OF M UNTIL A VALUE IS       */
    /* FOUND LESS THAN A(L), THEN SWAP            */
    DO WHILE (A(M)>A(L));
      M=M-1;
      END;
    IF M>N THEN DO;
      TEMP=A(N);
      A(N)=A(M);
      A(M)=TEMP;
      N=N+1;
      M=M-1;
      END;
    END;
  END;
/* THE ARRAY IS NOW PARTITIONED SO SWAP A(M)      */
/* AND A(L)                                       */
TEMP=A(M);
A(M)=A(L);
A(L)=TEMP;
```

FIGURE 4.10 - PARTITIONING PROGRAM

```
A(1)=10
A(2)= 5
A(3)= 4
A(4)=19
A(5)=18
A(6)= 1
A(7)= 3
A(8)=15
```
(4.8)

Assume that L=1 and U=8 to start. The pointers M and N are set to point at their starting positions:

```
A(1)=10
A(2)= 5<--N
A(3)= 4
A(4)=19
A(5)=18
A(6)= 1
A(7)= 3
A(8)=15<--M
```
(4.9)

N is moved until a value is found larger than A(L), and M is moved until a value is found smaller than A(L).

```
A(1)=10
A(2)= 5
A(3)= 4
A(4)=19<--N
A(5)=18
A(6)= 1
A(7)= 3<--M
A(8)=15
```
(4.10)

These values are swapped and the process is repeated.

```
A(1)=10
A(2)= 5
A(3)= 4
A(4)= 3
A(5)=18<--N
A(6)= 1<--M
A(7)=19
A(8)=15
```
(4.11)

Again the values are swapped and the process is repeated.

```
A(1)=10
A(2)= 5
A(3)= 4
A(4)= 3                                                    (4.12)
A(5)= 1<--M
A(6)=18<--N
A(7)=19
A(8)=15
```

It is now found that M and N have passed each other in the array. The array has been partitioned, so A(L) and A(M) are swapped to complete the process.

```
A(1)= 1
A(2)= 5     VALUES ≤ A(M)
A(3)= 4
A(4)= 3
---------                                                  (4.13)
A(5)=10     A(M)
---------
A(6)=18
A(7)=19     VALUES > A(M)
A(8)=15
```

The process can now be repeated on A(1) to A(4) and A(6) to A(8). Because there are now two subarrays to be partitioned and only one can be done at a time, the other subarray must be remembered. This is done by storing the lower and upper positions of the subarrays in two arrays LOWER and UPPER. Whenever a subarray is partitioned the positions of the new subarrays are placed in the arrays LOWER and UPPER. After partitioning a subarray, the positions of the next subarray are removed from LOWER and UPPER. This process is repeated until all subarrays are sorted.

Figure 4.11 shows the statements used to store the subarrays and recall them when needed. The variable CNT keeps a count of the number of subarrays yet to be partitioned.

```
/* PLACE UPPER AND LOWER BOUNDS ON ARRAY INTO    */
/* 'UPPER' AND 'LOWER' AND INITIALIZE CNT        */
LOWER(1)=LBOUND(A,1);
UPPER(1)=HBOUND(A,1);
CNT=1;
/* LOOP TILL NO MORE SUBARRAYS ARE LEFT          */
DO WHILE (CNT¬=0);
  /* IF THE SUBARRAY CONTAINS 2 OR LESS ELEMENTS*/
  /* THEN SORT IT AND DELETE IT FROM THE LIST.   */
  /* OTHERWISE, PARTITION IT INTO TWO SUBARRAYS  */
  /* AND ADD THEM TO THE LIST                    */
  L=LOWER(CNT);
  U=UPPER(CNT);
  CNT=CNT-1;
  IF U=L+1 THEN IF A(L)>A(U) THEN DO;
    TEMP=A(L);
    A(L)=A(U);
    A(U)=TEMP;
    END;
  IF U>L+1 THEN DO;
    /* AT THIS POINT THE STATEMENTS OF FIGURE    */
    /* 4.10 SHOULD BE INSERTED TO PARTITION THE  */
    /* SUBARRAY                                  */

    /* ADD NEW SUBARRAYS TO LIST                 */
    CNT=CNT+1;
    LOWER(CNT)=L;
    UPPER(CNT)=M-1;
    CNT=CNT+1;
    LOWER(CNT)=M+1;
    UPPER(CNT)=U;
    END;
  END;
```

FIGURE 4.11 - QUICKSORT PROGRAM

4.7 USE OF POINTER ARRAYS

One problem occurs when any of the above sorts are used with the student structure - to swap two student structures involves swapping every item within the structure. If the sort is being used with a very large structure, much more time is going to be spent swapping information than getting it into order. By using a pointer array this problem can be eliminated.

The pointer array is a list of numbers that point to the data in its proper order. For example -

```
A(1)=10      B(1)= 1
A(2)=23      B(2)= 2
A(3)= 5      B(3)= 3                    (4.14)
A(4)=75      B(4)= 4
A(5)=60      B(5)= 5
```

The array A holds the information that must be sorted. The array B points to the array A in its current order. After sorting the arrays will be -

```
A(1)=10      B(1)= 3
A(2)=23      B(2)= 1
A(3)= 5      B(3)= 2                    (4.15)
A(4)=75      B(4)= 5
A(5)=60      B(5)= 4
```

B(1) points to the smallest value in array A, the value of A(3), while B(5) points to the largest value, A(4). Note that the array A has not been changed. The array B has been changed so that it now points to the information in the A array in its sorted order. This method should be used whenever large amounts of information must be swapped in sorting.

CHAPTER 5 CHARACTER STRING MANIPULATION

PL/I was designed to use the best programming features that were currently available in FORTRAN and COBOL. All that has been discussed so far in this book are features that were taken from FORTRAN. COBOL's contributing features were those of character string manipulation.

5.1 DECLARING CHARACTER VARIABLES

In declaring character strings two items are necessary - the attribute CHAR or CHARACTER and the number of characters that the variable should hold, e.g.

```
DECLARE A CHAR(10);                                    (5.1)
```

This statement declares a variable A which holds 10 characters. An additional attribute of VARYING may be specified. This allows the variable to contain only as many characters as are placed into it.

```
DECLARE B CHAR(10) VARYING;                            (5.2)
```

5.2 ASSIGNMENT STATEMENTS

If the variables A and B declared above are now both assigned to hold the characters 'DOG', the variable A will contain the characters 'DOG' followed by seven blanks while the variable B will only contain the characters 'DOG'. To make these assignments the characters 'DOG' are placed inside single quotation marks as the following assignment statements show -

```
A = 'DOG';
B = 'DOG';                                          (5.3)
```

Character strings are always placed into variables left justified so that the variable A contains the characters 'DOG' in the three leftmost characters and they are followed by the seven blanks. Note that blanks are significant when assignments are made to variables that have been declared varying, e.g.

```
B = 'DOG ';                                         (5.4)
```

results in the variable B containing the characters 'DOG' followed by a single blank.

Because the variable A has not been declared as being varying, the variable A must always contain 10 characters. When the characters 'DOG' are assigned to it, the remaining character positions are filled or padded with blanks.

If A or B had instead been assigned the characters 'ABCDEFGHIJK' another problem arises.

```
A = 'ABCDEFGHIJK';
B = 'ABCDEFGHIJK';                                  (5.5)
```

A and B cannot hold more than 10 characters and the assignment statements try to place 11 characters in each variable. Because room is only available for 10 characters, the first 10 characters are placed into the variables with the remaining character 'K' being thrown away.

5.3 CONCATENATION

As with numbers, it would be quite useful to add characters to character strings that already exist. The concatenation operation is used to perform this. Suppose that the variables A and B from above each contained the characters 'DOG' and the characters 'CAT' are to be added to each. This can be expressed as shown below.

```
A = A || 'CAT';
B = B || 'CAT';                                    (5.6)
```

The two vertical bars, ||, are the concatenation operator. The resulting string in B will be 'DOGCAT' while A will not be changed. No change will occur in the variable A because it already contains 10 characters and concatenating on 3 more results in a character string 13 characters long. This string is then assigned to the variable A resulting in the last three characters or 'CAT' being truncated or thrown away.

5.4 BUILT-IN FUNCTIONS

Many character string built-in functions are available in PL/C and PL/I including INDEX, SUBSTR, LENGTH, and VERIFY.

LENGTH is used to determine the length of a character string. If the variables A and B from above each contain the characters 'DOG' the following would be observed -

1) LENGTH(A) would be 10, because the variable A was declared to always contain 10 characters.
2) LENGTH(B) would be 3, because B has been declared to be varying in length and contains the characters 'DOG'.

INDEX is used to locate the positions of certain substrings in character strings. Two arguments are used in this function. The first is the character string that is being looked in, while the second is the character string that is being looked for. For example, (assuming that B contains 'DOG') -

```
K= INDEX(E,'O');                                    (5.7)
```

would result in a value of 2 being assigned to K because 'O' is the second character in the variable B.

```
J= INDEX(B,'OG');                                   (5.8)
```

would also result in a value of 2 being assigned to J because the string 'OG' starts in the second character of B.

```
L= INDEX(B,'X');                                    (5.9)
```

will result in zero being assigned to L because 'X' does not occur anywhere in the variable B.

The VERIFY function is very similar to the INDEX function. Two arguments are used. The first is the character string that is being looked into while the second string consists of the characters that should be ignored in the first string. The value returned is the position of the first character in the first string that does not occur in the second string.

```
I= VERIFY (B,'OG');                                 (5.10)
```

will result in a value of 1 being assigned to I because the character 'D' is not in the second string.

```
M= VERIFY(B,'OD');                                  (5.11)
```

will result in a value of 3 being assigned to M and

```
N= VERIFY(B,'ODG');                                 (5.12)
```

will result in zero being assigned to N because all of the characters in the variable B are contained in the second string.

The SUBSTR function is used to obtain substrings or parts of character strings. Either two or three arguments are used. The first is always the character string that is being examined. The second argument is the position of the first character in the substring. The third argument, if given, is the number of characters to be in the substring. If a third argument is not given, the resulting substring will consist of all remaining characters in the string. The following examples show the results of using the SUBSTR function in various ways.

SUBSTR('AECD',2,1) results in 'B'

SUBSTR('ABCD',2) results in 'BCD'

SUBSTR('ABCD',3,2) results in 'CD'

SUBSTR('ABCD',3) results in 'CD'

SUBSTR('AECD',4,2) results in an error because an attempt is made to examine more characters than are in the original string.

SUBSTR can also be used as a pseudo-variable to specify where characters should be placed in a variable. Suppose that the variable A, declared to be CHAR(10), contains the characters 'DOG'.

SUBSTR(A,4,3)='CAT'; would cause the variable A to become 'DOGCAT '

SUBSTR(A,3)='DOG'; would cause A to become 'DODOG '

SUBSTR(A,5)='ABCDEFGH'; would cause A to become 'DOG ABCDEF'

SUBSTR(A,11,2)='AB'; would result in an error because A contains only 10 characters.

5.5 CHARACTER STRING CONDITIONS

Character strings can also be used in testing conditions. The following examples show the results of testing some conditions.

'A' = 'A ' true, the first string is padded with blanks on the right

' A' = 'A ' false, blanks are significant in comparisons

'ABC' > 'ABB' true, comparison is made character

by character in the string from the left until two characters are found diferent. The character 'C' is considered to be greater than the character 'B'.

'FRANK CARL' < 'FRANK, CARL' true

'Z' < '4' true

'1' < '4' true

' ' < 'A' true

The order of the characters in size from largest to smallest is given below with the smallest given on the left -

.<(+|&$*);¬-/,%_>?:#@'=ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

with a blank being less than '.'.

5.6 NULL STRING

The null string or string of characters containing no characters can be assigned tc variables. Assuming A is CHAR(10) and B is CHAR(10) VARYING -

```
A = '';
B = '';                                                    (5.13)
```

will result in A being set to all blanks and B being set to the null string. The length of a null string is zero.

CHAPTER 6 PROCEDURES

There are two major reasons why procedures are used in programs. Many problems are too large and complex to solve as a whole. These problems are broken into smaller problems which are hopefully easier to solve than the entire original problem. Each of these smaller problems can be solved and debugged separately and then combined to form the finished program that solves the original problem. This can also speed the debugging time for while one small problem is being debugged the other small problems can also be debugged. This technique is used quite often in industry. A skilled programmer known as a system's analyst is assigned a problem. He examines the problem and breaks it into a number of smaller problems giving each of his programmers one of these problems to solve. Each programmer then concentrates on his program and works until an acceptable programmed solution is reached. The system's analyst then combines all of these small programs into one large program to achieve the desired overall solution.

The other reason for using procedures is that programmers are basically lazy. If a program solution requires some operation to be performed many times in the program, a programmer will write up this operation only once and make it a procedure. Whenever the operation should be performed, the procedure is then referenced and executed. This saves the programmer from having to write and punch this commonly used operation many times within the program. The general form of a PROCEDURE is given in Figure 6.1. Comments should always procede a procedure. They should give a clear explanation of what the procedure is attempting to do.

The ENTRY_NAME is an identifier that contains at most seven

alphanumeric characters. This identifier is used in referencing

```
/* Comments About the Procedure */
ENTRY_NAME:PROCDURE{(List of Parameters)}
       {RETURNS(Return Attributes)}
       {RECURSIVE};
Declaration of Parameters and Local Variables
Statements to be Executed
END {ENTRY_NAME};
```

FIGURE 6.1 - GENERAL FORM OF A PROCEDURE

the procedure (i.e., causing the procedure to be executed). The ENTRY_NAME is followed by a colon and the word PROCEDURE or PROC.

An optional list of parameters follow the word PROCEDURE.

Parameters are identifiers that are replaced by variables when the procedure is referenced. They can be considered to be dummy names that will be replaced by real variables or values when the procedure is referenced.

A procedure can be used in one of two ways: by being CALLed or used as a function. If the procedure is used as a function procedure, the type of value that will be given back or returned by the procedure must be specified. This is done with the RETURNS clause. Inside the parentheses is placed the type of value that will be returned by the procedure (e.g., FIXED DECIMAL, CHAR(80), FIXED BINARY(10,2)). The value specified in the procedure to be returned will then be converted to this type and returned to where the procedure is referenced. A CALLED procedure does not use the RETURNS clause.

Finally, the procedure can be specified as being RECURSIVE. A RECURSIVE procedure is one that references itself or causes itself to be executed.

Following the procedure statement are normally DECLARE statements to declare the parameters and whatever temporary variables will be needed by the procedure. The temporary or local

variables are created when the procedure is referenced, used during the procedure's execution, and destroyed when the procedure ends its execution. They are declared in the same way variables are declared in the main program with one exception: the parameters may be used to give the dimensions of a temporary array or length of a temporary character string. In this way a temporary array or character string can be set up to a variable size, a size depending on the parameter's value.

In declaring a parameter care should be taken to make sure that the attributes given to the parameter are the same as the corresponding variable's (the variable that is associated with the parameter when then procedure is referenced). If the parameter does not have the same attributes, difficulties can arise.

Following the DECLAREs are the statements to be executed by the procedure. The procedure is ended with an END statement.

6.1 PLACEMENT OF PROCEDURES IN A PROGRAM

Procedures can be placed in two places in relation to the main procedure: internal to the main procedure or external to it. Below is an example showing several internal and external procedures.

```
A:PROC OPTIONS(MAIN);
    :
    :
  B:PROC;
     :
    END B;
    :
    :
  C:PROC;
     :
    C1:PROC;
        :
       END C1;
     :
    END C;
    :
```

```
      :
    END A;                                          (6.1)
$PROCESS
 D:PROC;
      :
      :
    D1:PROC;
        :
      END D1;
      :
    END D;
$PROCESS
 E:PROC;
    :
   END E;
```

Procedures B, C, and C1 are all internal procedures of A. C1 is also internal to C. Procedures D and E are external procedures. Procedure D also has an internal procedure D1. Note that the external procedures follow the end of the main procedure A and that they are proceded by a $PROCESS card. The $PROCESS cards are special cards used to inform the PL/C compliler that what follows is an external procedure. The $PROCESS cards must be typed starting in column 1 with no blanks between the letters. The $PROCESS card must also be used to signify external procedures when using PL/I.

6.2 CALLED PROCEDURES

Called procedures are referenced or invoked by using a CALL statement. the called procedure of Figure 6.1 might be invoked by using the following CALL statement -

```
CALL SWAP(A,B);                                     (6.2)
```

In the CALL statement the word CALL is followed by the name of the procedure to be executed. Following the name of the procedure are the variables that are to be associated with the parameters specified by the procedure. The variable A will be associated with the parameter X and the variable B will be associated with the parameter Y. Expressions and constants can be

used in place of the variables in a CALL statement.

```
/* PROCEDURE TO SWAP TWO VALUES */
SWAP:PROCEDURE(X,Y);
  DCL (X,Y) FIXED DEC;
  T=X;
  X=Y;
  Y=T;
  RETURN;
  END;
```

FIGURE 6.1 - SAMPLE CALLED PROCEDURE

The RETURN statement is used to state that the procedure is over and execution should now proceed from the place the procedure was called.

6.3 FUNCTION PROCEDURES

Function procedures are referenced or invoked by using the procedure name in an assignment statement or in some expression to be evaluated. The function procedure of Figure 6.2 can be invoked by the three statements shown below -

```
    :
DCL B(17) FLOAT DEC;
DCL A(10) FIXED DEC;
    :
PUT LIST(LARGE(B,5));                                    (6.3)
    :
Q=LARGE(A,J);
    :
P=A(LARGE(B,K+3));
    :
```

```
/* FIND THE LARGEST NUMBER IN AN ARRAY    */
LARGE:PROC(A,N) RETURNS(FIXED DEC);
  /* A IS THE ARRAY                       */
  /* N IS THE NUMBER OF ARRAY ELEMENTS TO */
  /*   BE SEARCHED                        */
  DCL (A(*),N,T) FIXED DEC;
  T=A(1);
  DO J=2 TO N;
    IF A(J)>T THEN T=A(J);
    END;
  RETURN(T);
  END;
```

FIGURE 6.2 - SAMPLE FUNCTION PROCEDURE

The use of a function procedure looks very similar to a reference to an array element. In the first useage of the function procedure, an array B is specified and a number '5'. The procedure will be executed by looking thru the first five numbers in the array B and returning the value of the largest to be output by the PUT LIST statement. In the second useage, Q will be given the value of the largest number in the array A thru the Jth number. In the last useage, P will be given the value of a number in the array A. The postion in the array A will be the largest number in the first 'K plus three' positions of array B.

Because the size of the array that will be used with this function is not known, a size can not be given to the parameter A in the procedure when the parameter is declared. Instead a '*' is given for the size. This means that the parameter A is a one dimensional array and that the procedure should accept whatever size the corresponding variable is. The '*' must also be used for the length of a character string parameter. For example, the following DECLARE statement from a procedure declares a parameter that is a two dimensional array of character strings -

```
DCL ST(*,*) CHARACTER(*);                                    (6.4)
```

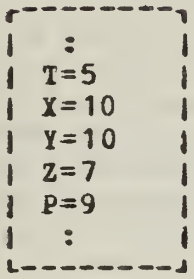
6.4 INTERNAL VS. EXTERNAL PROCEDURES

Internal and external procedures not only differ in their placement within the program, but also in the variables that are available for their use. Suppose the SWAP procedure, given in Figure 6.1 above, has been used with the main procedure shown in Figure 6.3 as an internal procedure.

```
TEST:PROC OPTIONS(MAIN);
       :
  DCL (T,Z,P) FIXED DEC;
  DCL (Y,X) FLOAT DEC;
       :
  T = 5;
  X,Y=10;
  Z = 7;
  P = 8;
       :
  CALL SWAP(P,X);
       :
  SWAP:PROC(X,Y);
         :
    END SWAP;
  END TEST;
```

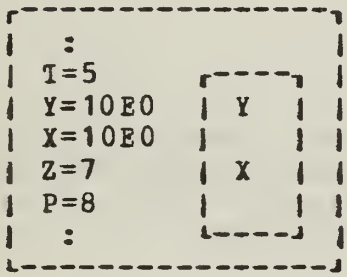
FIGURE 6.3 - MAIN PROCEDURE WITH A PROCEDURE CALL

When the program is executed, a block of memory is used to store the variables that have been declared in the main procedure. This block of memory might look as follows just before executing the CALL statement:



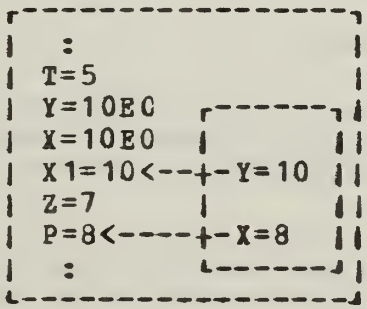
(6.5)

When the CALL statement is executed, a new block of memory must be set up to hold the parameters and local variables declared within SWAP. Because SWAP is an internal procedure, this block of memory will be created internal to the block of memory used by the main procedure.



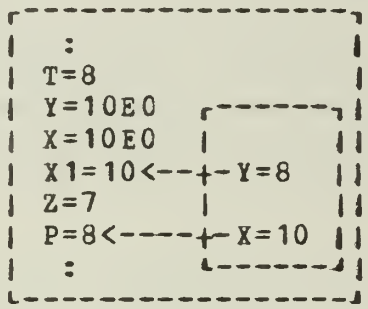
(6.6)

Next, the parameters are set so that they point to the variables that they are associated with. A check is made to make sure that the parameters and variables have the same attributes. If they do not, a dummy variable is set up to hold the value of the variable. This dummy variable's attributes are made to match those of the parameter's. The values that are in the variables are now transferred in to the parameters.



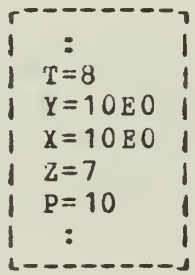
(6.7)

The procedure now starts to execute. The first statement says to set T equal to the value in X. The SWAP procedure looks through its list of parameters and variables for T and X and only is able to find X. Because the procedure is internal to another block of memory, a search is then made through those variables for T and a T is found. The value of T is set to that of X. X is set to the value of Y and Y to the value of T.



(6.8)

The RETURN statement is now executed. The values in paramters Y and X are transferred back to the variables that they point to. The parameters and any temporary variables are destroyed, leaving the following results:



(6.9)

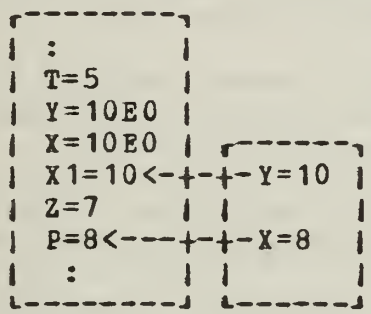
The important results of this example are:

1) The value of the parameter Y are placed in the temporary variable X1 when the RETURN is executed. X1 is then destroyed without transferring this value into X.

2) Because SWAP is internal to another procedure, when variables cannot be found in SWAP's block of memory a

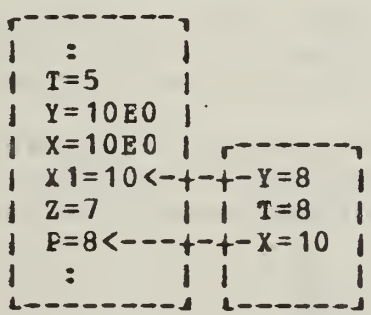
hunt is made through the next larger block. This hunt will continue until either the variable is found or there are no more containing blocks. If it is not found (i.e. there are no more containing blocks), the variable is set up as a local or internal variable to the current procedure's block of memory. In other words, if T had not existed in the main procedure, it would have been set up as a local variable inside the block of memory associated with SWAP. Because a T was found it was used, changing its value.

Suppose that SWAP had been used in the program as an external procedure instead of an internal procedure. When the CALL statement was executed the following would have resulted :



(6.10)

When the procedure tries to set T equal to X, a search is made through all of the internal variables and parameters and T is not found. This block of memory is not contained internal to any other so T is set up as a local variable and used. Just before executing the RETURN statement the values would be:



(6.11)

The RETURN statement is then executed resulting in

```
┌--------┐
|  :     |
| T=5    |
| Y=10E0 |
| X=10E0 |
| Z=7    |
| P=10   |
|  :     |
└--------┘
```

(6.12)

The major difference in the results is that the variable T in the main procedure was not changed using SWAP as an external procedure. The following observations should be noted:

1) The parameter and variable names declared in the procedure are in no way related to variables having the same names in the main or calling procedure.

2) Parameters are not really variables. They are used to point to or refernece the variables to be used.

3) The pointers from the parameters are set before execution of the procedure is started and cannot be changed until after the procedure is through executing.

4) External procedures must be given all values to be used from the main procedure through the parameter list. If the value is not 'passed' through the parameters it cannot be used.

5) Internal procedures know the global variables (variables in the larger blocks) so they need to be given as much or as little as the programmer wishes.

6) A CALLed procedure can change the value of the variables passed to the procedure. A function procedure can change these passed variables and also returns a calculated value.

6.5 NESTING

Procedures can occur inside procedures that are internal to others. Procedure calls or references can occur anywhere a normal statement can occur.

```
A: PROCEDURE OPTIONS(MAIN);
     :
  B:PROC;
     :
    C:PROC;
       :
    END C;
     :
    D:PROC;
       :                                          (6.13)
    END D;
     :
  END B;
     :
   AA:____
     :
  E:PROC;
     :
    BB:____
     :
  END E;
END A;
$PROCESS
F:PROC;
     :
   CC:____
     :
  G:PROC;
     :
    H:PROC;
       :
    END H;
     :
  END G;
     :
END F;
```

Statement AA is internal to the procedure A. From that statement the procedures B, E, and F can be called. From statement BB procedures E, B and F can be called. note that neither statement can call procedures C, D, G or H because they are contained within procedures and cannot be seen by the two statements. Similarly, statements CC can call procedures F and G but not H, or any of the procedures contained in A. This is because those procedures cannot be seen from statement CC.

Note that statement CC can call procedure F, the procedure that contains it. A procedure which calls itself is known as a recursive procedure and will be discussed later in this chapter.

6.6 BEGIN BLOCKS

Many times programs are written to handle a wide range of data. For example, a grading program might be desired by a university. This program should be able to compute the grade point averages of all the students enrolled in the university. Once the program has been written, it would seem logical to allow University High School (UNI) to also use this program to compute the grades of its students. A problem arises however. It would be very wasteful for the program to always be run for a maximum of 40,000 students. There are not 40,000 students at UNI so much space would be wasted and next semester the university might have more that 40,000 students, sc more space would be needed. It would be much more logical to allow the user to specify exactly how many grade point averages are going to be computed. BEGIN blocks allow this to be done. The general form of a BEGIN block is shown in Figure 6.4. BEGIN blocks can be considered as a cross between PROCEDUREs and the simple DO statement. When encountered during execution, the statements within the BEGIN block are executed as statements are within a DO block. Procedures are ignored when encountered and are only executed when referenced. PROCEDUREs and BEGIN blocks allow the declaring of internal variables which DO blocks do not allow. The execution of a BEGIN block is very similar to the execution of a PROCEDURE. First, the variables that are declared are set up and assigned initial values if any are given. The statements in the block are then executed as they normally would be. When the END statement of the block is enccuntered all variables that were declared are

destroyed and execution continues at the statement following the

```
{Label:} BEGIN;
   Declare Statements
   Executable Statements
   END {Label};
```

FIGURE 6.4 - GENERAL FORM OF A BEGIN BLOCK

end of the block.

The grade problem might be arranged as shown in Figure 6.5.

Suppose that the university not only wanted to compute the grade point averages, but also figure the class schedules of all the students. It would be very wasteful to keep all course information for both semesters in memory at the same time, so two BEGIN blocks might be used. The first would be used for computing the grade point averages. A structure would be declared to hold the needed information and the calculations performed. This BEGIN block would then be ended and a second BEGIN block could be used to set up the class schedules. When the first BEGIN block is ended, the memory used by it is free to be used again and can be used by the second BEGIN block. Using this method problems that would require more memory than is available on the machine can be solved.

6.7 RECURSION

A recursive procedure is one which calls or invokes itself. Many problems can be solved using recursion and some can only be solved by the use of recursion. A recursive procedure consists of two states. The first is the ending condition or case where the answer is known. The second is the recursive state where the procedure calls itself with a simpler problem that the original

problem.

```
GRDS:PROC OPTIONS(MAIN);
  GET LIST(N);
  /* READ THE NUMBER OF STUDENT RECORDS TO BE */
  /* PROCESSED                                */
  L:BEGIN;
    DCL 1 STUDENT(N),
          2 ST_NAME CHAR(30),
          2 COURSES(10),
            3 GRADES CHAR(1),
            3 CRS_NAME CHAR(10),
          2 GPA FIXED BINARY(20,10);
       :
       :
    /* STMTS TO READ INFORMATION, DO THE      */
    /* REQUIRED CALCULATIONS AND OUTPUT THE   */
    /* THE RESULTS                            */
       :
    END L;
  END GRDS;
```

FIGURE 6.5 - GENERAL LAYOUT OF GRADING PROGRAM

A good example of recursion is the computing of N factorial which is defined as

(N) (N-1) (N-2) ... (2) (1) (6.14)

The ending condition for this problem is when N has a value of 1. The factorial of N at this point is simply 1. The recursive step is then that N factorial is equal to N times (N-1) factorial for all other numbers. Before this value can be computed a value must be obtained for (N-1) factorial. So the procedure is repeated. It is easy to see that finding (N-1) factorial will be done in the same way as finding N factorial, but the problem is easier - the number whose factorial is being found is one less than before. Applying this procedure again and again will eventually reach the point of finding the factorial of 1, which is know. The steps for finding N factorial can be represented as follows:

$$N \text{ factorial} = \begin{cases} 1 \text{ if } N<2 \\ N * (N-1) \text{ factorial otherwise} \end{cases} \qquad (6.15)$$

Writing these steps as a procedure, obtains the procedure of Figure 6.6.

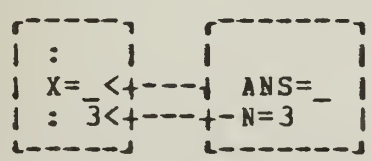
```
FAC:PROC(N) RECURSIVE RETURNS(FIXED DEC);
  DECLARE (N,ANS) FIXED DEC;
  IF N<2 THEN RETURN(1);
  ANS=N*FAC(N-1);
  RETURN(ANS);
  END;
```

FIGURE 6.6 - RECURSIVE PROCEDURE TO COMPUTE N FACTORIAL

Suppose that FAC was an external procedure and was referenced by

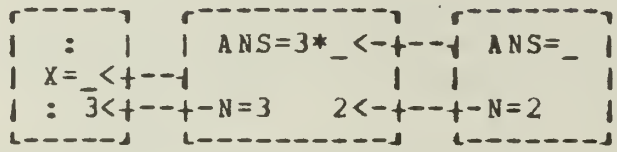
```
X = FAC(3);                                          (6.16)
```

External to the block of memory for the main procedure, a new block of memory would be set up for FAC. A pointer would be set from this block to X to show where FAC should return its value.



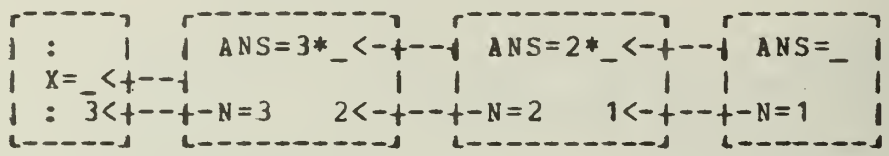
(6.17)

The statements are then executed. The check on N being less than 2 is made and found to be false so ANS is given the value of N times FAC(N-1), or



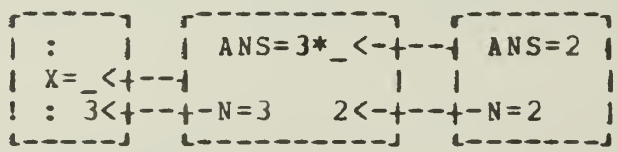
(6.18)

In this second call of FAC the test for N being less than 2 is made and fails sc ANS is set to N times FAC(N-1) giving



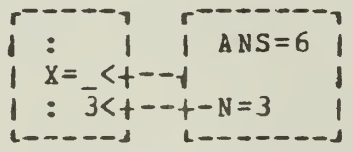
(6.19)

The check is made to see if N is less than 2 and it is, so a value of 1 is returned to the previous call of FAC. This causes ANS to be set to 2.



(6.20)

The value of ANS is then returned to the place where FAC had been called with N=2. This causes the previous value of ANS to get the value 6.



(6.21)

This value of 6 is then returned and given to X giving X the value of 3 factorial.

(6.22)

To reverse a character string recursively is quite easy. First, the ending condition is found. If a character string has a length of 1 or less characters then the reverse of that character string is the original string. The reverse of any other character string is just the reverse of that character string with the first character removed, and the first character concatenated on the end or

```
                 ┌- ST IF LENGTH(ST)<=1
REVERSE(ST) -    |                                    (6.23)
                 └- REVERSE(SUBSTR(ST,2))||SUBSTR(ST,1,1)
                    OTHERWISE
```

Written as a procedure this yields the program of Figure 6.7.

```
REVERSE:PROC(ST) RECURSIVE RETURNS(CHAR(256) VARYING);
  DCL(T) CHAR(256) VARYING, ST CHAR(*) VARYING;
  IF LENGTH(ST)<=1 THEN RETURN(ST);
  T=REVERSE(SUBSTR(S,2))||SUBSTR(S,1,1);
  RETURN(T);
  END;
```

FIGURE 6.7 - RECURSIVE PROCEDURE TO REVERSE A CHARACTER STRING

CHAPTER 7 ADVANCED TECHNIQUES

Discussed in this chapter are several advanced techniques that are normally not used by beginning programmers.

7.1 THE ON STATEMENT

The ON statement has the two general forms given in Figure 7.1. The option, SNAP, results in a printing of a list of all

```
FORM 1 - ON 'CONDITION' {SNAP} SYSTEM;

FORM 2 - ON 'CONDITION' {SNAP} ON-UNIT
```

FIGURE 7.1 - GENERAL FORMS OF THE ON STATEMENT

procedures that are active at the time the condition is raised. SYSTEM means to take the standard system action of that condition. The on-unit can be a single unlabeled statement except for a DO, IF, RETURN, or another ON statement. It can be an

unlabeled BEGIN statement.

The execution of an ON statement does not result in the on-unit being executed. The on-unit is associated with the specified condition and when the specified condition is raised the on-unit is then executed. Commonly used conditions are:

1) CHECK(NAME {,NAME}) - NAME is the name of a simple variable, array, structure, or label. The CHECK condition is raised whenever an assignment is made to one of the specified names that is a simple variable, element of an array or element of a structure or just before execution of the statement identified with a specified label name. The CHECK condition is used mainly to obtain debug output and results in the identifier name being printed with any value that it might have. Upon the termination of the on-unit, if given, execution returns to the point where the condition was raised and the program continues.

2) ENDFILE(FILE-NAME) - FILE-NAME is the name of an input file (e.g. SYSIN). This condition is raised when a GET statement is executed and no more data remains in the specified file. Upon completion of the on-unit, the program is continued to be executed with the statement immediately following the GET statement.

3) FLOW - This condition is raised at any point in the program when statements are not executed in their normal numeric order. This occurs in the execution of CALL, DO, GO TO, RETURN, and IF statements and in function references. Upon completion of the on-unit the program continues where the condition was raised.

7.2 LABEL VARIABLES

LABEL variables are used when a GO TO statement must cause a transfer to several different places in the program. LABEL variables are declared by

```
DECLARE VARIABLE-NAME LABEL;                    (7.1)
```

The specified variable-name can then be assigned any label name within the program. An example of its use is shown in Figure 7.2.

```
  :
  :
DECLARE X LABEL;
  :
IF A>B THEN X=HERE;
       ELSE X=THERE;
  :
  :
GO TO X;
  :
  :
HERE:;
  :
  :
THERE:;
  :
  :
```

FIGURE 7.2 - SAMPLE PROGRAM USING LABEL VARIABLES

X is assigned the label HERE or THERE depending on the results of the IF statement. The GO to statement will result in transfer to either HERE or THERE depending on which label X has been assigned. Label arrays can also be declared and initial values of labels can be given in the DECLARE statement if desired.

7.3 CONTROLLED VARIABLES

CONTROLLED variables cannot be used in PL/C. They are part of the PL/I language. A CONTROLLED variable is declared as shown in Figure 7.3. A sample DECLARE statement might be -

```
DECLARE X FIXED DECIMAL CONTROLLED;                    (7.2)
```

```
DECLARE VARIABLE-NAME 'ATTRIBUTE' CONTROLLED;
```

FIGURE 7.3 - GENERAL FORM OF THE DECLARATION OF A CONTROLLED VARIABLE

A CONTROLLED variable does not exist in the program until the programmer specifies so by using an ALLOCATE statement -

```
ALLOCATE X;                                            (7.3)
```

This causes the variable X to be set up in memory. Values can now be given to the variable.

```
X=1;                                                   (7.4)
```

If another ALLOCATE statement occurs for X, the current value of X is "pushed down" and saved. A new X is created and can now be assigned values.

```
ALLOCATE X;                                            (7.5)
X=10;
```

This X variable is continued to be used until a FREE statement is encountered which destroys the current variable.

```
FREE X;                                                (7.6)
```

This results in the variable X which contains the value of 10 being destroyed. The previous value of X which was "pushed down" is now used, so X has a value of 1. Freeing a CONTROLLED variable that does not exist (i.e., has not been allocated) will result in an execution error that will terminate execution of the program.

7.4 POINTER AND BASED VARIABLES

These variables also can only be used in PL/I. POINTER variables are used in linking items together in a specified order. POINTER variables are declared as shown in Figure 7.4.

```
DECLARE VARIABLE-NAME POINTER;
```

FIGURE 7.4 - DECLARATION OF POINTER VARIABLES

POINTER variables can only be assigned machine addresses of BASED variables. BASED variables are declared as shown in Figure 7.5.

```
DECLARE VARIABLE-NAME1 BASED(VARIABLE-NAME2);
```

FIGURE 7.5 - DECLARATION OF BASED VARIABLES

BASED variables must be freed and allocated like controlled variables. When a BASED variable is allocated it is referenced by using the BASE variable. For example, suppose the following variables have been declared -

```
DECLARE Q1 POINTER;
DECLARE 1 AA BASED(Q),                                    (7.7)
          2 VAL FIXED DECIMAL,
          2 LINK POINTER;
```

When AA is allocated, Q is assigned its address.

```
ALLOCATE AA;                                              (7.8)
```

results in Q being given the machine address of the structure AA. To refer to the values in AA the following form must be used -

```
Q->VAL = 10;                                          (7.9)
Q->LINK = NULL;
```

This causes the current variable VAL to be assigned 10 and the value of LINK to be set to NULL. NULL is an address that does not occur in the machine and is one that can be used for tests. 'Q->' is used to tell the computer to use the locations of VAL and LINK that are based or pointed to by Q.

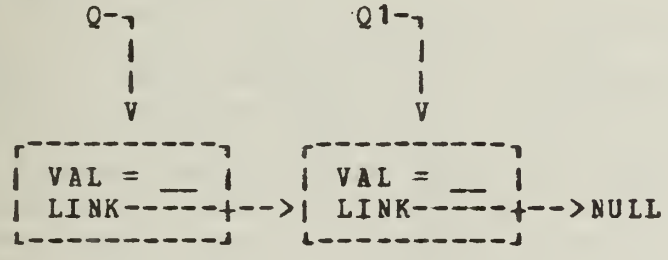
If AA is allocated again, Q is changed to point tc the new AA. The old value of Q is lost forever. To keep that value it must be temporarily stored in some POINTER variable, say Q1.

```
Q1 = Q;                                               (7.10)
ALLOCATE AA;
```

Q now points to the new AA and Q1 to the previous AA.

```
Q->LINK = Q1;                                         (7.11)
```

will cause the twc structures being linked together so that the following appears in memory



(7.12)

To FREE a BASED variable, the specific variable being freed must be specified.

```
FREE Q->AA;                                           (7.13)
```

will free the latest AA structure while -

```
FREE Q1->AA;                                          (7.14)
```

will free the first AA structure.

7.5 CHARACTER TO NUMBER CONVERSION AND VICE VERSA

Automatic conversion from numeric characters to numbers and numbers to characters is not allowed in PL/C as it is in PL/I. In order to do this conversion a special form of the GET and PUT statement must be used. This form uses the STRING option. Figure 7.6 shows a GET and PUT statement using this option.

```
GET STRING(C) EDIT(I)(X(2),F(3));

PUT STRING(D) EDIT(I)(F(7,4));
```

FIGURE 7.6 - SAMPLE GET AND PUT EDIT STATEMENTS USING THE STRING OPTION

The statements of Figure 7.6 are executed just like any other GET and PUT statements, except rather than GETting the information from data cards the information is input from the specified character string, C, and rather than PUTting the information to the line printer the information is output to the specified character string, D. Assume that C and D have been declared to be CHAR(10), and both contain 'xy 247p9rt'. The GET statement says to go into C, skipping the first two characters, then converting the next three characters into a fixed number and assigning the value to the variable I. The execution of the statement results in I being assigned a value of 24. The PUT statement will take this value, convert it into 24.0000 and assign it to D giving D a value of '24.0000 '.

Several important things should be noted. The STRING option can be used with any form of the GET or PUT statement. SKIP, COLUMN, PAGE, and LINE which have no meaning in a character string cannot be used as format specifications or options. A SUBSTR can be used as the specified character string if desired.

CHAPTER 8 EXPLANATIONS OF THE PL/C ERROR MESSAGES

The following is an explanation of the compile errors that can occur when running programs in PL/C. Explanations of the execution errors are not given because they tend to be self explanatory.

The error messages appear as they will on the printout with the exception of the following words which will be replaced by variable information:

IDEN - a variable or label name will be printed
STRING - a character string will be printed
NUMBER - a fixed or floating number will be printed
RTN - the name of a subroutine will be printed
LINE - a statement number will be printed
ATTRIBUTE - an attribute will be printed

A line beginning PROGRAM CHECK or COMPILER ERROR indicates a problem in the PL/C compiler and not a user error (although it most often occurs in response to some user error). Please notify a T.A. or consultant if such errors occur so that Cornell can be notified and the problem remedied.

These explanations are intended to assist beginning PL/C programmers in debugging programs. The corrections which PL/C makes to errors will often result in new errors which are totally unrelated to the actual bug in the problem. For this reason the corrections described in this chapter and the actual corrections made by PL/C should be treated with skepticism. Do not assume

that PL/C knows what is wrong and accept the corrections generated. Each error and correction should be carefully examined to determine if the action taken by PL/C is acceptable and truly represents what the programmer desired. If the corrections do not represent the programmer's desires a careful examination should be made to determine why the error did occur.

The following example are intended to show how irrelevant error messages can be generated. These examples are typical of common errors generated by beginning programmers.

EXAMPLE 1

In examining Figure 8.1, two things should be immediately noticed. First, the statement

```
MAXNBR=-10;                                    (8.1)
```

does not have a statement number on the left. This implies that the statement has been ignored. Second, all of the error messages refer to statements 2 and 3.

Look at the error messages for statement 2. These errors seem to be referring to some additional variable that is being declared. The second error states that an attempt was made to declare a variable twice. The correction that is made to this error is to replace the variable with one that PL/C makes up, namely $V001$. The final corrected statement contains this variable and all others from the original statements, but does not contain the attributes FIXED DEC which were used. This implies that the attributes have also been ignored. Using this information, the location of the problem is determined to be the third line of the DECLARE statement. In examining the third line of the DECLARE, the error is quickly discovered to be that the comment is ended improperly with /* rather than */.

In discovering this error the meaning of all of the error messages becomes obvious. Because the comment on the third line of the DECLARE statement was not ended on that line, the compiler continued the comment until the next */ was found. This was

```
$PL/C

*OPTIONS IN EFFECT*     TIME=(0,10),PAGES=20,LINES=2000,NOATR,NOXREF,FLAGW,NOBNDRY,NOCMNTS,SORMGIN=(2,72,1),
*OPTIONS IN EFFECT*     ERRORS=(50,50),TABSIZE=4324,UDEF,SOURCE,OPLIST,NOCMPRS,NOHDRPG,AUXIO=10000,LINECT=60,NOALIST,
*OPTIONS IN EFFECT*     MCALL,MTEXT,DUMP=(S,F,L,E,U,R),DUMPE=(S,F,L,E,U,R),DUMPT=(S,F,L,E,U,R)


 /* COMPUTE THE MAXIMUM OF NON-NEGATIVE NUMBERS   */                          PL/C-R7.1--00 10/06/75 22:21 PAGE   1

  STMT LEVEL NEST BLOCK MLVL  SOURCE TEXT

                             /* COMPUTE THE MAXIMUM OF NON-NEGATIVE NUMBERS   */
                             /* DUMMY -1 ADDED FOR STOPPING TEST     */
     1                       FINDMAX:PROCEDURE OPTIONS(MAIN);

     2    1          1       DECLARE (NUMBER,            /* THE CURRENT NUMBER */
                                 MAXNBR,                 /* MAXIMUM VALUE SO FAR */
                                 COUNT)                  /* NBR OF NUMBERS SO FAR /*
                                 FIXED DEC;

                             MAXNBR=-10;        /* INITIAL VALUE LESS THAN ALL */
                                                /* POSSIBLE DATA VALUES */
      ERROR IN STMT    2  MISSING COMMA IN COLUMN 2 (SY06)
     3    1          1       COUNT=0;
      ERROR IN STMT    2  MULTIPLE DECLARATION IN COLUMN 2 (SY27)
      ERROR IN STMT    2  MISSING SEMI-COLON (OR MISUSE OF RESERVED WORD) (SY08)
          FOR STMT    2 PL/C USES  DECLARE (NUMBER,MAXNBR,COUNT),$V0001$;

      ERROR IN STMT    3  IMPROPER ELEMENT IN COLUMN 7 (SY16)
      ERROR IN STMT    3  IMPROPER ELEMENT IN COLUMN 8 (SY16)

          FOR STMT    3 PL/C USES  ;

     4    1          1       GET LIST (NUMBER);

     5    1          1       DO WHILE (NUMBER ¬= -1);
     6    1    1     1           COUNT=COUNT+1;
     7    1    1     1           IF NUMBER>MAXNBR THEN MAXNBR=NUMBER;
     9    1    1     1           GET LIST (NUMBER);
    10    1    1     1           END;

    11    1          1       PUT LIST('NUMBER OF VALUES =', COUNT);
    12    1          1       PUT SKIP LIST('MAXIMUM VALUE =', MAXNBR);
    13    1          1       END FINDMAX;

ERRORS/WARNINGS DETECTED DURING CODE GENERATION:

     WARNING: NO FILE SPECIFIED. SYSIN/SYSPRINT ASSUMED. (CG0C)
```

FIGURE 8.1 - SAMPLE PROGRAM 1 WITH ERRORS

found three lines below, following

```
/* INITIAL VALUE LESS THAN ALL                    (8.2)
```

Included within this long comment were the attributes of the DECLARE statement and the first assignment statement -

```
MAXNBR=-10;                                       (8.3)
```

The statement

```
COUNT=0;                                          (8.4)
```

was then found. Because the DECLARE statement had not yet been ended with a semicolon, the compiler thought that COUNT was something being declared, so a comma was inserted. The compiler then discovered that COUNT had previously been declared so it printed a message to state this. COUNT was then destroyed and $V001$ was substituted in its place. The compiler continued looking for a semicolon and next found an equal sign. Because an equal sign cannot occur in a DECLARE statement, a semicolon was inserted to end the DECLARE statement. All that remained of statement 3 was

```
=0;                                               (8.5)
```

Because a statement cannot start with an equal sign, it was thrown away, leaving

```
0;                                                (8.6)
```

A statement, also, cannot start with a numeric constant so the zero was thrown away, leaving just the semicolon.

EXAMPLE 2

Figure 8.2 exhibits a similar problem with comments. Note that statement numbers are missing from statement 3 to the end of the program. The first error message states that the comment was never ended, but a */ occurs at the end of all comments in the program. The reason this error has been given is that the / in the last comment is punched in column 73 on the card. PL/C by default reads only columns 2 through 72, so the / was never seen.

There are three possible ways of insuring such problems do not occur. First, a very careful check should always be made of programs before running them. Look carefully for comments starting in column 1 (the /* in this case will signify the end of the program) and statements or comments that extend past column 72. Second, specify BNDRY as an option on the PL/C card (see Appendix A). This will not allow comments and strings to extend across card boundaries. Finally, specify SORMGIN=(2,80,1) as an option on the $PL/C card. This will allow punching of comments and statements anywhere on the card except in column 1. This last solution is not always the best one to use, because many users prefer to punch sequencing numbers in columns 73 to 80. These numbers order the cards numerically in case the deck should be dropped so the cards can quickly and easily be placed back in order.

These two examples, while only examining errors dealing with comments, show the difficulty that exists in debugging programs. Debugging programs is a skill that must be developed and that can only be done by programming and puzzling over the error messages which result. Hopefully, the following explanations will decrease the puzzling time.

```
$PL/C

*OPTIONS IN EFFECT*     TIME=(0,10),PAGES=20,LINES=2000,NOATR,NOXREF,FLAGW,NOBNORY,NOCMNTS,SORMGIN=(2,72,1),
*OPTIONS IN EFFECT*     ERRORS=(50,50),TABSIZE=4324,UDEF,SOURCE,OPLIST,NOCMPRS,NOHDRPG,AUXIO=10000,LINECT=60,NOALIST,
*OPTIONS IN EFFECT*     MCALL,MTEXT,DUMP=(S,F,L,E,U,R),DUMPE=(S,F,L,E,U,R),DUMPT=(S,F,L,E,U,R)

 /* COMPUTE THE MAXIMUM OF NON-NEGATIVE NUMBERS   */                          PL/C-R7.1--00 10/06/75 22:27 PAGE   1

  STMT LEVEL NEST BLOCK MLVL  SOURCE TEXT

                             /* COMPUTE THE MAXIMUM OF NON-NEGATIVE NUMBERS   */
                             /* DUMMY -1 ADDED FOR STOPPING TEST     */
     1                       FINDMAX:PROCEDURE OPTIONS(MAIN);

     2    1          1       DECLARE (NUMBER,            /* THE CURRENT NUMBER */
                                     MAXNBR,             /* MAXIMUM VALUE SO FAR */
                                     COUNT)              /* NBR OF NUMBERS SO FAR */
                                     FIXED DEC;

     3    1          1       MAXNBR = -10;     /* INITIAL VALUE LESS THAN ALL POSSIBLE DATA VALUES */
                             COUNT=0;

                             GET LIST (NUMBER);

                             DO WHILE (NUMBER ¬= -1);
                                 COUNT=COUNT+1;
                                 IF NUMBER>MAXNBR THEN MAXNBR=NUMBER;
                                 GET LIST (NUMBER);
                                 END;

                             PUT LIST('NUMBER OF VALUES =', COUNT);
                             PUT SKIP LIST('MAXIMUM VALUE =', MAXNBR);
                             END FINDMAX;
    ERROR IN STMT    4  MISSING */ BEFORE END OF FILE OR CONTROL CARD IN COLUMN 20 (SY56)
    ERROR IN STMT    4  MISSING END (SYOE)
        FOR STMT     4 PL/C USES  END;
```

FIGURE 8.2 - SAMPLE PROGRAM 2 WITH ERRORS

8.1 VARIABLE PREFIX ERRORS

NUMBER MESSAGE

E2 ERROR LIMIT EXCEEDED
(The program has exceeded the maximum number of errors specified by the user or defaulted by the system. Possible corrections include
1. Eliminate errors within program.
2. Increase the error limit by changing the 'ERRORS=(C,R)' option on the $PL/C card.)

E3 LINE LIMIT EXCEEDED
(The program has printed more lines than specified by the user or allowed by the system. Possible corrections include
1. If running EXPRESS
   A. Suppress the printing of the program by using the NOSOURCE/SOURCE option on the $PL/C and/or $PROCESS cards. This will allow continued running on EXPRESS.
   B. Start running on HASP. Check the ID cards and remove the 'LINES=350' specification if used.
2. If running HASP
   A. Increase the 'LINES=N' limit on the ID cards. Default limit is 1000.
   B. Increase the 'LINES=N' limit on the $PL/C card. Default limit is 2000.)

E4 PAGE LIMIT EXCEEDED
(The program has exceeded the maximum number of pages specified by the user or defaulted by the system. Possible corrections include
1. Increase the 'PAGES=N' limit on the $PL/C card.
2. Suppress the printing of the program by using the 'NOSOURCE/SOURCE' option on the $PL/C and/or $PROCESS cards.)

E5 TIME LIMIT EXCEEDED
(The program has exceeded the maximum time specified by the user or defaulted by the system. Possible corrections include
1. Correct all other errors in the program and run again.
2. Check for a possible infinite loop by adding debug output statements to the program.
3. Increase 'TIME=(M,S)' limit on the ID and $PL/C cards and run on HASP.)

E6 TIME LIMIT EXCEEDED - PROBABLE COMPILER LOOP
(See T.A. or a consultant.)

E7 I/O ERROR *STRING*
(A system error has occurred. The *STRING* given is a SYNADAF error message. Possible corrections include
1. See T.A. or a consultant.
2. See the appropriate IBM system manual.)

E8 UNABLE TO PROCESS INCLUDE COMMAND
(Possible corrections include
1. Make certain the data set name specified is spelled correctly.
2. Check with T.A. to make sure the data is in the computer.
3. If running on HASP, make certain the program has a
```
//PLCLIB DD DSN=SYS1.EXPRESS.UIXAUX,DISP=SHR
```
card after the
```
// EXEC PLC
```
card.)

E9 SYMBOL-TABLE OVERFLOW. USE LARGER REGION OR INCREASE TABSIZE
(See T.A. or a consultant.)

EA STRING TOO LONG FOR LINE IN ABOVE MESSAGE (OR COMPILER ERROR)
(Correct the error above this one. If this error re-occurs see T.A. or a consultant.)

8.2 SY OR MD PREFIX ERRORS

NUMBER MESSAGE

00 MISSPELLED KEYWORD
(Apparent misspelling of one of the reserved words.)

01 EXTRA (
(PL/C will delete it. For example
PUT LIST((A);)

02 MISSING (
(PL/C will add it. For example
PUT LIST A);)

03 EXTRA)
(PL/C will delete it. For example
X=C+D);)

04 MISSING)
(PL/C will add it. For example
X=(C+D;)

05 EXTRA COMMA
(PL/C will delete it. For example
GET LIST (A,,B);)

```
06    MISSING COMMA
         ( PL/C will add it. For example
            GET LIST(A B);
         will become
            GET LIST(A,B);
         or
            PUT LIST('IT'S A DOG');
         will become
            PUT LIST('IT',S,A,DOG,');
         where
            PUT LIST('IT''S A DOG');
         was wanted. )

07    EXTRA SEMI-COLON
         ( PL/C will delete it. For example
            IF X=1; THEN ...
         will become
            IF X=1 THEN ...          )

08    MISSING SEMI-COLON (OR MISUSE OF RESERVED WORD)
         (  A  semi-colon is supplied by PL/C. Frequently this
         error is caused by attempting to use a reserved  word
         as  a  variable name. If this is the case, change the
         variable name so it is not a reserved word.  Examples
         include
            PUT LIST(A,ON);
         where ON is a reserved word or
            X=A DO I=1 TO 3;
         where the semi-colon is missing )

09    MISSING :
         ( PL/C will add it. For example
            X DO I=1 TO 10;
         where
            X:DO I=1 TO 10;
         was intended. )

0A    MISSING =
         ( PL/C will add it. For example
            A B+C;
         where
            A=B+C;
         was intended. )
```

0B IMPROPER ATTRIBUTE ON PARAMETER

(A parameter declared in this DECLARE statement does not match the declaration of its corresponding variable. For example

```
Z:PROCEDURE OPTIONS(MAIN);
DCL X FIXED DECIMAL;
   :
   :
P=Q(X);
   :
   :
   Q:PROCEDURE(N) RETURNS(FIXED DECIMAL);
   DCL N CHAR(*);
      :
      :
   END Q;
   :
   :
END Z;
```

where N and X do not match in their declaration.)

0C INEFFECTIVE IF

(A pointless IF statement has been used. The THEN clause is null and no ELSE clause exists. This might be caused by an extra semi-colon. For example

```
IF A=B THEN; GO TO X;
```

where

```
IF A=B THEN GO TO X;
```

was intended.)

0D IMPROPER ENTRY/RETURNS ATTRIBUTE

0E MISSING END

(A $PROCESS, $END or the end of the program was encountered with a block still open. Carefully check the program to be certain that all DO loops, BEGIN blocks, and PROCEDUREs have END statements. It is possible that all the needed END statements are present with this error being created by a correction to another error.)

```
0F    MISSING KEYWORD
        ( PL/C inserts what it considers to be the proper
        keyword. Check this statement carefully to be certain
        that the correction is what was intended. For example
           GET (A,B,C);
        would be corrected to
           GET LIST(A,B,C);
        with this error. )

10    INCOMPLETE EXPRESSION
        ( A variable has most likely been left out of an
        expression. For example
           A=B+C* ;
        which was intended to be
           A=B+C*D;
        but D was not punched on the card. )

11    MISSING EXPRESSION
        ( Possibly the right hand side of an assignment
        statement is missing. For example
           A= ;     )

12    MISSING VARIABLE
        ( Could be caused by not having left hand side to an
        assignment statement. For example
           =C+D;    )

13    MISSING ARGUMENT, 1 SUPPLIED
        ( An attempt was made to use a procedure or builtin
        function without giving enough arguments (variables).
        An example might be the builtin function INDEX which
        requires two arguments. An attempt was made to use
        the function with only one argument. For example
           X=INDEX(P);
        which should have been
           X=INDEX(P,I);      )

14    EMPTY LIST
        ( An attempt was made to use the GET or PUT statement
        with no variables or expressions in the list. For
        example
           GET LIST();
        this could have been caused by correction of other
        errors. )
```

15 IMPROPER NOT

(The ¬ cannot be used as a binary operator so ¬= is substituted. For example

```
IF A¬B THEN ...
```

will become

```
IF A¬=B THEN ...
```

or

```
IF ¬A=B THEN ...
```

where

```
IF ¬(A=B) THEN ...
```

was intended.)

16 IMPROPER ELEMENT

(A variable or operator has occurred in an improper place. This error occurs frequently due to corrections of other errors. Make a good attempt to find what has caused this error but if unable to find it, correct all other errors and hope it disappears.)

17 IMPROPER SYNTAX, TRANSLATION SUSPENDED

(PL/C got so lost in trying to correct this statement that it gave up. It replaced the statement by a NULL statement and ignored everything that followed up to the next reserved word or semi-colon.)

18 INCONSISTENT OPTION, STATEMENT DELETED

(An attempt was made to use some option in the statement and it was improperly used. The statement was deleted and replaced by a NULL statement. Reread the explanation of the option in the book and try to correct it. If the first correction fails then see a grader or T.A. so they can explain what is wrong.)

19 NOT ENOUGH CORE, TRY LARGER REGION

(The program is too large to be run with the amount of space requested by the user or the default value. See T.A., a grader or a consultant to see how to increase the region before making another run.)

1A NESTING TOO DEEP

(The nesting level in the program exceeded the capacity of PL/C. The limits are

1. 12 for IF statements (count all THEN and ELSE clauses within the statement to determine the level).
2. 11 for PROCEDURE s, BEGIN s and DO s.
3. 6 for factors in a DECLARE statement.
4. 14 for parentheses in expressions.)

1B UNACCESSIBLE STATEMENT

(This statement cannot be reached during execution. For example

```
GO TO X;
A=B+C;
```

The second statement is unaccessible. Either the GO TO statement is placed incorrectly or a label is missing from the second statement.)

1C MISSING MAIN PROC

(A statement similar to the following is missing from the start of the program -

```
NAME:PROCEDURE OPTIONS(MAIN);
```

)

1D MISSING PROCEDURE STATEMENT

(This error is normally the result of too many END statements being in the program. Match all DO, PROCEDURE and BEGIN statements with the corresponding END statement to make certain they are matched properly. Ocassionally this error results from other errors, so correction of them will eliminate this error.)

1E MISSING $PROCESS (OR EXTRA END)

(Either a $PROCESS card has been omitted between external procedures or an extra END statement exists (see error 1D above).)

1F MISPLACED ENTRY STATEMENT
(Check to make certain that the entry statement is not contained in a BEGIN block or in an iterative DO loop.)

20 IMPROPER OPTIONS(S)
(An improper or invalid option has been used on a $PL/C, $PROCESS or $OPTIONS card. Possible corrections include
1. See if the option is legal.
2. Check the spelling and use of the option.
3. See T.A. or a consultant.)

21 IMPROPER FORMAT ITEM
(The current statement is a PUT or GET EDIT statement, some illegal format specification is used in the second set of parentheses.)

22 IMPROPER I/O PHRASE
(Something is very wrong with the current GET or PUT statement. Check for proper use of parenthesis and keywords. This error can be generated if an attempt is made to use GET or PUT as an identifier.)

23 IMPROPER TO PHRASE
(Something is very wrong with the TO clause of the current DO statement or an attempt has been made to use TO as an identifier.)

24 IMPROPER BY PHRASE
(Something is very wrong with the BY clause of the current DO statement or an attempt has been made to use BY as an identifier.)

25 IMPROPER WHILE PHRASE
(Something is very wrong with the WHILE clause of the current DO statement (there should be parenthesis around the condition) or an attempt has been made to use WHILE as an identifier.)

26 IMPROPER SPECIFICATION
(The iteration specification of the DO statement is in error. Only TO, BY or WHILE clauses should be used.)

27 MULTIPLE DECLARATION
(An attempt has been made to redeclare an identifier. PL/C will create a dummy identifier to replace this one.)

28 IMPROPER *ATTRIBUTE* ATTRIBUTE FOR *IDEN*
(The *ATTRIBUTE* indicated cannot be applied to this identifier (*IDEN*) due to previous attributes. This error is generated in a DECLARE statement and might be caused by PL/C corrections to other errors in that statement.)

29 IMPROPER FACTORING
(Most likely caused by mismatched parenthesis in a DECLARE statement.)

2A IMPROPER DIMENSION
(An illegal dimension has been specified for an identifier in a DECLARE statement. Remember that variable dimensions can only be given and used in BEGIN blocks and internal or external procedures. This error can be caused by corrections that PL/C has made previously in the statement.)

2B IMPROPER PRECISION
(The precision value that is being applied to a variable is illegal. This error can be generated by corrections PL/C makes to other errors.)

2C IMPROPER SCALE
(The scale value that is being applied to a variable is illegal. This error can be generated by corrections PL/C makes to other errors.)

2D IMPROPER VARYING ATTRIBUTE
(An attempt was made to apply the VARYING attribute to a variable that is not CHARACTER or BIT. This error can be generated by corrections PL/C makes to other errors.)

2E IMPROPER FILE-NAME
(The identifier in the file phrase is not a valid file-name. See T.A. or a consultant.)

2F EXTERNAL NAME TOO LONG
(A maximum of 7 characters is allowed. See T.A. or a consultant.)

30 IMPROPER INIT ATTRIBUTE
(Attempt has been made to initialize a variable to some value incompatible with what it was declared as. Such as initializing a fixed decimal variable to a character string. This error can be generated by corrections PL/C make to other errors.)

31 IMPROPER STRUCTURE LEVEL
(The structure being declared is constructed improperly or a stray integer has occurred in the DECLARE statement. This error can be generated by corrections PL/C makes to other errors.)

32 IMPROPER ATTRIBUTE IN STRUCTURE
(Major and minor structure names cannot have type attributes. This error can be generated by corrections PL/C makes to other errors.)

33 TOO MANY IDENTIFIERS
(PL/C allows a maximum of 88 identifiers in a single factor or structure.)

34 IMPROPER THEN OR ELSE
(A THEN or ELSE clause has occurred without an IF statement. This can be caused by attempting to use THEN or ELSE as identifiers or forgetting DO groups as follows

```
      :
      :

   IF A=B THEN P=Q;
               R=S;
          ELSE A=T;
      :
      :
```

This error would result with the ELSE clause placed as shown. What was most likely intended was (where a DO group was missing)

```
      :
      :
   IF A=B THEN DO;
               P=Q;
```

```
                    R=S;
                    END;
               ELSE A=T;
          :
          :
```

or (where the second assignment statement was misplaced)

```
          :
          :
     R=S;
     IF A=B THEN P=Q;
          ELSE A=T;
          :
          :
```

This error is also generated by corrections PL/C makes to other errors.)

35 IMPROPER THEN OR ELSE UNIT

(The statement used as the THEN or ELSE clause is not allowed. If it is a THEN clause a NULL statement will be substituted. An ELSE is just eliminated. Statements that cannot be used as THEN or ELSE clauses include END, PROCEDURE, and DECLARE.)

36 MISSING THEN

(A THEN is missing from the IF statement. This can be caused as follows

```
   IF A=B; THEN GO TO X;
```

where

```
   IF A=B THEN GO TO X;
```

was intended.)

37 IMPROPER CHECK OR NOCHECK

(The prefix has been applied to a statement other than BEGIN or PROCEDURE. The prefix will be deleted.)

38 IMPROPER PREFIX ORDER

(This is a warning that the specified order will result in an error if run with PL/I-F, but it will execute properly under PL/C.)

39 EXTRA LABEL

(This occurs when more than one label is given to a PROCEDURE statement. For example

```
   X:Y:PROCEDURE OPTIONS(MAIN);      )
```

3A IMPROPER LABEL
(The label on the statement is illegal. It might start with a numeric character or contain too many characters.)

3B MISSING LABEL OR ENTRY NAME
(A label has not been placed on a PROCEDURE or ENTRY statement. PL/C will generate a label for this statement.)

3C IMPROPER ON-CONDITION
(This error is often the result of starting an assignment statement with a left parenthesis. For example
(X=Y;)

3D IMPROPER ON-UNIT
(This statement is not allowed with an ON condition. PL/C will supply a BEGIN and END statement to make it legal.)

3E IMPROPER SPACE
('NO' was improperly separated from the rest of the keyword. The space will be removed. For example
NO CHECK
will become
NOCHECK)

3F PL/I FEATURE NOT IN PL/C
(An attempt was made to use a feature of PL/I which has not yet been implemented in PL/C.)

40 FEATURES INCOMPATIBLE WITH PL/I-F HAVE BEEN USED
(A PL/C feature has been used that cannot be used with PL/I-F. This is a warning to the user.)

41 INCOMPATIBLE OPTION
(An option that can be used with the given statement has been used improperly or an attempt has been made to use something that is not an option for the statement.)

42 MISSING OPTION
(A required option has not been given.)

43 IMPROPER LOGICAL UNIT
(This is generated under DOS only. It means that an improper logical unit in MEDIUM option of the ENVIRONMENT attribute has been used.)

44 MISSING DECIMAL INTEGER

45 NON-* BOUND/LENGTH FIELD
(A * must be given for the subscript bound or string length. An expression cannot be used. This normally occurs when declaring a string or array parameter in a procedure.)

46 DECLARATION FOR ENTRY *IDEN* DOES NOT AGREE WITH CORRESPONDING PROCEDURE OR ENTRY POINT

47 PROCEDURE *IDEN* IS NOT PRESENT
(A reference to the specified procedure has been made but the procedure has not been found. This can be caused by a simple misspelling or by an improperly placed procedure.)

48 *-LENGTH NOT ALLOWED. 256 USED
(The variable being declared cannot have a * for the string length. 256 has been used in place of the *.)

49 TOO MANY DIGITS IN EXPONENT
(A maximum of two digits can be used in an exponent.)

4A ILLEGAL EXPONENT
(An illegal character has occurred following the 'E'.)

4B ILLEGAL BINARY NUMBER
(A character other than 1 or 0 has occurred in a binary number. The number will be treated as a decimal number.)

4C ILLEGAL USE OF COLUMN 1 ON CARD

4D */ NOT IN COMMENT
(The */ will be ignored. Check to see if the /* accompanying it has been mistyped or is missing.)

4E NAME>31 CHARACTERS
(An identifier has been used that is too long.)

4F ILLEGAL CHARACTER
(Retype the card. Some column on the card has most likely been punched with two characters or a character has been used that is not legal in PL/C.)

50 STRING CONSTANT RUNS ACROSS CARD BOUNDARY
(A ' has been left out somewhere or occurs beyond column 72.)

51 IMBEDDED BLANK(S) IN OPERATOR
(The blanks will be ignored. For example
* *
will become
**)

52 COMMENT RUNS OVER CARD BOUNDARY
(A */ will be supplied and all kinds of errors will result if the comment continues on the next card without a starting /*. Make sure comments end on each card or use the NOBOUNDARY option on the $PL/C card.)

53 2 DECIMAL POINTS IN NUMBER
(The number will be ended before the second decimal point.)

54 EXPONENT RUNS ACROSS CARD BOUNDARY
(Numbers must be typed entirely on one card. The exponent will be ignored.)

55 SPACE MISSING BETWEEN NUMBER AND LETTER
(A space will be inserted. This could be due to a missing operator or starting an identifier with a numeric character.)

56 MISSING */ BEFORE END OF FILE OR CONTROL CARD
(Make certain a */ is not beyond column 72.)

57 INVALID BIT STRING
(A character other than a 0 or 1 has occurred in a bit string.)

58 MISSING QUOTE BEFORE END OF FILE OR CONTROL CARD
(Make certain that the quote has not been left out or has not been placed beyond column 72 on a card.)

59 STRING LENGTH>255
(The character or bit string contains too many characters.)

5A MISPLACED *MEND CARD
(A $MEND card is missing following a MACRO.)

5B ERROR STACK OVERFLOW - MESSAGE(S) LOST
(Correct as many errors as possible and run again. If this error re-occurs see I.A. or a consultant.)

5C TABSIZE TOO LARGE. DEFAULT USED
(Decrease the size of the TABSIZE option on the $PL/C card or increase the core size specified for the program.)

5D OPTION(S) NOT ALLOWED AT THIS INSTALLATION
(A option specified on a $PL/C, $PROCESS or $OPTIONS card is prohibited.)

5E TOO MANY SIGNIFICANT DIGITS, 16 USED

5F TOO MANY SIGNIFICANT DIGITS, 53 USED

60 EXPONENT TOO LARGE

61 NUMBER TOO SMALL

62 NUMBER TOO LARGE/SMALL, *NUMBER* USED
(Either 0 or 10**75 will be used, as appropriate.)

63 MISSING *MEND BEFORE END OF FILE OR CONTROL CARD

64 MISSING MACRO NAME

65 MISSING %; BEFORE END OF FILE OR CONTROL CARD

66 MACRO NAME ILLEGAL OR ALREADY IN USE

67 MISSING PARAMETER NAME IN MACRO DEFINITION

68 MACRO PARAMETER NAME >31 CHARACTERS. FIRST 31 USED

69 TOO MANY MACRO PARAMETERS. LIST TRUNCATED

6A MACRO PARAMETER NAME APPEARS TWICE IN LIST

6B SYMBOL TABLE AREA OVERFLOW. INCREASE CORE AVAILABLE
(See T.A. or grader.)

6C ILLEGAL CHARACTER(S) ON CARD. BLANK(S) USED
(Retype card because some columns have more than one character typed in them or a character has been used that is not legal in PL/C.)

6D MARCO EXPANSICN CAUSES REPRINTING OF ABOVE LINE
(Warning that the above line has been printed twice.)

6E DYNAMIC CORE CVERFLOW DURING MACRO EXPANSION.
INCREASE REGION
(See T.A. or grader.)

6F SCANNING OF AEOVE LINE TERMINATES IN COLUMN *NUMBER*
(If unable to figure out why see T.A. or grader.)

70 COMPILER ERROR -- ILLEGAL INTERNAL MACRO PARM ID
(see T.A. or a consultant.)

71 MISSING (IN MACRO CALL

72 MACRO ARGUMENT >256 CHARACTERS. FIRST 256 USED

73 TCO FEW ARGUMENTS IN MACRO CALL. NULL STRING(S) SUPPLIED

74 MISSING COMMA IN MACRO CALL

75 MISSING) IN MACRO CALL

76 END OF FILE OR CONTROL CARD WITHIN MACRO CALL

8.3 SM PREFIX ERRORS

NUMBER	MESSAGE
40	VARIABLE NOT PERMITTED (A constant must be used in this context.)
41	WRONG TYPE FOR EXPRESSION (Expression types are ARITHMETIC, STRING, LABEL or FILE and the wrong one has been used here.)
42	WRONG STRUCTURE OR DIMENSIONALITY FOR EXPRESSION (A simple (scalar) variable is needed where an array or matching structure has been used.)
43	ILLEGAL SUBSCRIPTING (Subscripts not allowed in this context. For example with a GET DATA statement.)
44	ILLEGAL USE OF PSEUDO-VARIABLES
45	NAME NEEDED
46	ENTRY-NAME NEEDED (A CALL must have an entry (procedure) name.)
47	NO STRUCTURE APPEARED
48	STRUCTURES DO NOT MATCH (Attempted to set two structures equal to each other with an assignment statement and the structures are not declared identically.)

49 FUNCTION ARGUMENTS MISSING
(Attempting to use a function without supplying values to be used in evaluating the function.)

4A OPERAND OF BINARY OPERATOR *STRING* HAS IMPROPER TYPE
(This error can occur when attempting to add a bit or character string.)

4B OPERANDS OF BINARY OPERATOR *STRING* DISAGREE IN TYPE, STRUCTURE OR DIMENSIONALITY
(This error can occur when attempting to add two arrays or structures that differ in size or add a character string to a number.)

4C OPERAND OF UNARY OPERATOR *STRING* HAS IMPROPER TYPE
(This error can occur when attempting to use a + or - with a character or bit string.)

4D SUBSCRIPT *NUMBER* OF *IDEN* NOT NUMERIC
(The specified subscript might be a character or bit string.)

4E *IDEN* HAS TOO MANY SUBSCRIPTS. SUBSCRIPT LIST DELETED
(The specified identifier has been used with more subscripts than it was declared to have.)

4F *IDEN* HAS TOO FEW SUBSCRIPTS. SUBSCRIPT LIST DELETED
(The specified identifier has been used with fewer subscripts than it was declared to have.)

50 NAME NEVER DECLARED, OR AMBIGUOUSLY QUALIFIED

51 SUBSCRIPT *NUMBER* OF *IDEN* NOT SCALAR
(Attempting to use an entire array for the specified subscript.)

52 *IDEN* HAS TOO MANY ARGUMENTS. FUNCTION REFERENCE DELETED
(The specified function has been referenced with too many arguments. For example
SIN(X,Y)
where the SIN function can only have one argument, i.e. SIN(X))

53 ARGUMENT *NUMBER* OF FUNCTION *IDEN* DISAGREES WITH CORRESPONDING PARAMETER
(This might occur when attempting to compute the SIN of a character string .)

54 *IDEN* HAS TOO FEW ARGUMENTS. FUNCTION REFERENCE DELETED
(See SM52)

55 ARGUMENT *NUMBER* OF FUNCTION *IDEN* WAS *. ILLEGAL ARGUMENT

56 TABLE OVERFLOW. STATEMENT DELETED
(See T.A. cr grader.)

57 TABLE OVERFLOW. STATEMENT DELETED
(See T.A. or grader.)

58 TABLE OVERFLOW. STATEMENT DELETED
(See T.A. or grader.)

59 TABLE OVERFLOW. STATEMENT DELETED
(See T.A. or grader.)

5A *IDEN* HAS WRONG # OF SUBSCRIPTS
(Occurs when attempting to set two structures equal to each other and they differ in size.)

5B MISMATCHED DIMENSIONALITY
(Attempting to set two arrays equal to each other, but they differ in size.)

5C ILLEGAL LABEL VARIABLE *IDEN*
(Attempting to use a subscripted variable as a label when it has not been declared. Correct by declaring the specified variable as being a LABEL. This could also occur when PL/C is attempting to correct other errors and messes up.)

5D ILLEGAL ASSIGNMENT TARGET
(Something illegal is to the left of the equal sign.)

5E ASSIGNMENT SOURCE INCOMPATIBLE WITH TARGET
(Can occur when attempting to set a character string variable tc a numeric value, etc.)

5F MAJOR STRUCTURE NAME NEEDED

60 DEFAULT ATTRIBUTES FOR ENTRY NAME *IDEN* CONFLICT WITH RETURNS OPTION IN STMT *LINE*
(Should be a warning message not an error. Ignore.)

61 *IDEN* IS ASSUMED A VARIABLE, NOT A BUILT-IN FUNCTION
(The specified identifier has the same name as a built-in function. This message just notifies that the built-in function cannot be used in this program.)

8.4 XR PREFIX ERRORS

NUMBER	MESSAGE
62	NOT ENOUGH CORE FOR CROSS-REFERENCE (See T.A. or grader.)
63	CROSS REFERENCE ABBREVIATED DUE TO LACK OF SPACE (See T.A. or grader.)
64	COMPILER ERROR IN XREF PHASE -- INVALID STATEMENT CODE (See T.A. or a consultant.)

8.5 CG PREFIX ERRORS

NUMBER MESSAGE

00 FORMAT WILL BE EXECUTED ONLY ONCE
(The format of the EDIT statement does not contain format items that would cause data to be transferred in or out. There are no A, B, C, E, F or R formats.)

01 CONSTANT BOUND, LENGTH,SUBSCRIPT OR ITERATION FACTOR EXCEEDS 32767 IN MAGNITUDE. 10 IS USED
(The value of a variable, length of a character or bit string, subscript of a variable or iteration (DO) variable value is too large. This is a PL/C restriction.)

02 WORKSPACE OVERFLOW IN STATEMENT PROCESSING
(The combined nesting of BEGIN and PROCEDURE blocks, iterative DO groups, and IF statements is too deep. Increasing the region size will not help. The nesting depth must be reduced.)

03 *IDEN* REQUIRES TOO MUCH SPACE. UPPER BOUND OF SUBSCRIPT. *NUMBER* IS SET TO LOWER BOUND
(More than 2**31 bytes of storage would be required for this variable as it is declared. Decrease the size of the array.)

04 PRIMARY DATA STORAGE AREA FOR BLOCK # *NUMBER* EXCEEDS SIZE LIMIT BY *NUMBER* BYTES.
(Try adding some BEGIN blocks just inside the block that is causing this error or see T.A..)

05 LENGTH ID *IDEN* (*NUMBER*) IS NOT IN PROPER RANGE. 80 IS USED.
(An attempt has been made to use the specified element of an array that is CHARACTER or BIT in type. The length of that element is <0 or >256.)

06 *IDEN* REQUIRES TOO MUCH SPACE. LOWER BOUND OF SUBSCRIPT. *NUMBER* IS SET TO ZERO
(Shift the array so the lower bound is closer to zero and re-submit.)

07 VALUE IN RETURN HAS ATTRIBUTES INCOMPATIBLE WITH ONE OR MORE ENTRY POINTS.
(This is a warning that execution error might be generated if this return is used. A character string might be returned where a number is expected.)

08 SEVERE ERRORS. EXECUTION DELETED.
(An error already mentioned has made it impossible to continue with the program.)

09 CONVERSION REQUIRED TO MATCH ARGUMENT *IDEN* OF *IDEN*
(This is a warning that PL/C will have to do conversion when executing the CALL to the procedure. Change the declaration of the specified variables so they are declared identically and this error does not occur, otherwise wrong answers might result.)

0A SCALAR ARGUMENT SUPPLIED TO AGGREGATE PARAMETER *IDEN* OF *IDEN*. (1:10 USED FOR ALL BOUNDS.)
(See T.A. or a consultant.)

0B WORKSPACE OVERFLOW IN EXPRESSION PROCESSING
(The program is attempting to execute a statement (the statement number is given in the next error) and could not because the expression is too complex. Simplify the expression and re-submit.)

0C NO FILE SPECIFIED. SYSIN/SYSPRINT ASSUMED.
(This is just a warning. Ignore it.)

0D *IDEN* IS A PARAMETER IN I/O LIST
(This is just a warning that PL/I-F will not accept this statement as written. Ignore it.)

0E BOTH FORMS OF INITIALIZATION USED FOR LABEL VARIABLE *IDEN*
(This is just a warning that PL/I-F will not accept this statement as written. ignore it.)

0F STORAGE CAPACITY IS EXCEEDED
(See T.A., grader or a consultant.)

10 ILLEGAL COMPLEX COMPARE. REAL PARTS WILL BE COMPARED
(A comparison cannot be made between the variables.)

11 *IDEN* IS ILLEGAL OPERAND IN INITIAL, LENGTH OR DIMENSION ATTRIBUTE OF STATIC VARIABLE. *CONSTANT* IS USED.
(The bounds, lengths or dimensions of an external or static variable must be a decimal constant.)

12 NON-CONSTANT OPERAND (*IDEN) IN INITIAL LENGTH OR DIMENSION ATTRIBUTE OF STATIC VARIABLE.
(This is a warning that PL/I-F will not accept this statement as written.)

13 PL/C BUILT-IN FUNCTION USED.
(A built-in function not included in PL/I-F has been used.)

14 ARGUMENT TO 'MAX' OR 'MIN' IS COMPLEX. REAL PART USED.

15 NO SCALE FACTOR ARGUMENT APPEARED. RESULT IS SET TO FLOAT.

16 UNNECESSARY SCALE FACTOR ARGUMENT APPEARED. RESULT IS SET FIXED.

17 ARGUMENT SHOULD BE CONSTANT. *NUMBER* IS USED.
(Certain arguments to the built-in functions must be decimal constants.)

18 ABS (*ARGUMENT*) >32767. *NUMBER* IS USED.
(Constant argument of built-in functions must be less than 32768 in magnitude.)

19 ARGUMENT SHOULD BE REAL. IMAGINARY PART IS USED.
(The second argument of complex built-in functions or pseudo-variables must be real.)

1A ILLEGAL COMPLEX ARGUMENT, REAL PART IS USED.
(Certain functions are defined for real arguments only.)

1B ILLEGAL ARGUMENT TO BUILT-IN FUNCTION. SHOULD BE REAL, FIXED DECIMAL CONSTANT.
(This is a warning that PL/I-F will not accept this statement as written.)

1C RESULT SCALE FACTOR= *NUMBER* >127 IN MAGNITUDE. RESULT SCALED INCORRECTLY TO 127*SIGN(*NUMBER*).
(The scale factor, Q, of the result of the expression will be out of the permitted bounds. PL/C has changed the value so execution can continue. The result from this expression will be wrong.)

1D PROGRAM MAY LOOP IF THIS FORMAT IS EXECUTED
(See error CG00.)

1E VARIABLE *IDEN* HAS * BOUND OR LENGTH FIELD. 10 IS USED.
(Only parameters can have * for a bound or length. Separate the parameters and variables into two DECLARE statements.)

1F PARAMETER *IDEN* HAS A NON-* BOUND OR LENGTH FIELD
(Parameters must have a * in this field. Separate the parameters and variables into two DECLARE statements.)

20 LOWER BOUND ON SUBSCRIPT *NUMBER* OF *IDEN* EXCEEDS UPPER BOUND. (0:10) IS USED
(Either the subscript bounds are reversed or a minus sign has been misplaced.)

21 SPECIFIED (*NUMBER*) TOO LARGE. MAX PRECISION IS USED
(31 will be used if the variable is FIXED BINARY; 15 will be used if the variable is FLOAT DECIMAL.)

22 ARITHMETIC/BOOLEAN CONVERSION INVOLVED.

23 TOO MANY ERRORS DURING COMPILATION. EXECUTION SUPPRESSED.
(Correct as many errors as possible and re-submit. If error re-occurs see T.A. or grader.)

24 COMPILER ERROR DURING CODE GENERATION. PROGRAM ABORTED
(See T.A. or consultant.)

APPENDIX A - $PL/C CARD OPTIONS

Listed below are options that are available to the user to specify. The default action, if one exists, is listed in parentheses. The options can be given in any combination, in any order, separated by commas and/or blanks on the $PL/C card. These options can also be used on $PROCESS cards, which must occur before every external procedure and $OPTIONS cards, which can be placed anywhere in the program to change the current options. If more options are to be specified than can fit on one card, subsequent cards can be used by placing a $ in column 1 and leaving columns 2 and 3 blank. Individual options cannot be split over a card boundary.

ATR, (NOATR)

Controls the printing of an alphabetical listing of identifiers giving the attributes assigned to each identifier.

ALIST, (NOALIST)

Controls the printing of an assembler listing of the generated object code.

AUXIO=n (can be used only on the $PL/C card)

Specifies the limits on the number of auxiliary input/output operations. The supplied default is 10000.

BNDRY, (NOBNDRY)

Controls whether comments and literals (strings) can be continued across card boundaries. NOBNDRY permits continuation.

CMNTS, CMNTS=(n1,n2,...), (NOCMNTS)

This allows comments which start with : to be considered as source statements. If parameters are given (1<=ni<=7) then the comments beginning with nI are also considered to be source text.

CMPRS, (NOCMPRS)

The source listing will be printed in a compressed form. Certain page ejects will be replaced by 3 line skips.

DUMP, (DUMP=(d1,d2,...)), NODUMP (can be used only on the $PL/C card)

Produces a post-mortem dump. The dump options are:

B - traceback of the blocks that were active at program termination.

S - causes printing of the final values of all scalar variables in the active blocks. This implies B.

A - causes printing of the final values of all arrays in the active blocks. This implies both S and B.

F - causes printing of a history of the last 16 transfers of control (times when the statements were not executed in normal numeric order).

L - causes printing of all the labels used in the program and the frequency in which they were encountered.

E - causes printing of all entry names used in the program and the frequency in which they were

called.

R - causes a printing of the statistics on the run (time, core usage, a auxiliary input/output operations, etc.).

U - causes a listing of the first five of fewer unread data cards.

Depth - is an integer giving the limit on the number of active blocks for B, S, and A dump options. If 0 is given, the depth is unlimited.

The supplied default dump options are (S,F,L,E,U,R).

DUMPE, (DUMPE=(d1,d2,...)), NODUMPE (can be used only on the $PL/C card)

This produces the post-mortem dump only if an error is enccuntered during execution. The supplied default DUMPE options are (S,F,L,E,U,R).

DUMPT, (DUMPT=(d1,d2,...)), NODUMPT (can be used only on the $PL/C card)

This produces the post-mortem dump only if execution is terminated by an error. The supplied default DUMPT options are (S,F,L,E,U,R).

ERRORS= (c,e) (can be used only on the $PL/C card)

Specifies the error limits of the program. If more than c errors occur during compilation, the program will not be executed. E errors during execution will be allowed before terminating the program. The default values are (50,50).

FLAGE, (FLAGW)

Controls the printing of warning messages. FLAGW will cause both warning and error messages to be printed while FLAGE will only print errors.

HDRPG, (NOHDRPG) (can be used only on the $PL/C card).

Causes the printing of a header/separator page before the program.

ID=' name ' (can be used only on the $PL/C card)

Up to 20 characters can be given inside the single quotation marks. It is used for identification in the program heading line.

LINES= n (can be used only on the $PL/C card).

Specifies the maximum number of lines to be printed by the program. The supplied default is n=2000. Note if this option is changed it must also be changed on the ID cards.

LINECNT= n

Specifies the number of lines to be printed on each page of the source listing. The default is 60 and a minimum of 21 can be specified.

(MCALL), NOMCALL

Causes all macro calls to be printed.

(MTEXT), NOMTEXT

Causes printing of all macro text expansions.

(OPLIST), NOOPLIST

Causes printing of all options in effect.

PAGES= n (can be used only on the $PL/C card).

Specifies a page limit on the printed output. Default limits are 10 for HASP and 4 for EXPRESS. If this is increased on HASP be sure to also increase the number of lines to be printed. This limit cannot be changed on EXPRESS.

SORMGIN- (s,e,c)

Controls the margins of the cards for the source program. S is the first column scanned and is given a default value of 2. E is the last column scanned and is given a default value of 72. C is the carriage control column and is given a default of 1.

(SOURCE), NOSOURCE

Controls the printing of the source program listing.

TIME= (m,s) (can be used only on the $PL/C card).

Specifies the upper limit on the total processing time (compilation plus execution). This limit is given in m minutes and s seconds. The default limits are (0,4) on EXPRESS and (0,10) on HASP. If the HASP value is increased, do not forget to increase the time specification on the /*ID cards.

TABSIZE= n (can be used only on the $PL/C card)

This determines the amount of PL/C region allocated to the symbol table. The number, n, is given in full words. The supplied default is 1/2 of the usable area.

(UDEF), NOUDEF (can be usedonly on the $PL/C card)

Indicates whether or not PL/C checks for the use of a variable before being assigned a value. NOUDEF suppresses this checking.

XREF, (NOXREF)

Control the printing of an alphabetic listing of identifiers, indicating where each is declared and used.

APPENDIX B - BUILT-IN FUNCTIONS

The built-in functions listed below are available for use in PL/C. All of the functions are also available for use in PL/I except RAND. This list consists of all of the commonly used function. For a more complete listing consult AN INTRODUCTION TO PROGRAMMING by Conway and Gries for the PL/C built-in functions and the "IBM Reference Manual" for the PL/I built-in functions.

ARITHMETIC FUNCTIONS:

ABS(X) - returns the absolute value of X.

CEIL(X) - returns the smallest integer that is greater than or equal to X. X cannot be complex.

COMPLEX(X,Y) - returns a complex number with a real part of X and imaginary part of Y.

CONJG(X) - returns a complex number which is the conjugate of the complex number X.

FLOOR(X) - returns the largest integer not greater than X.

IMAG(X) - returns the imaginary part of X.

MAX(X1,X2,...,XN) - returns the maximum value of the arguments X1, X2, ..., XN.

MIN(X1,X2,..,XN) - returns the minimum value of the arguments X1,

X2, ..., XM.

MOD(X,Y) - returns the remainder when X is divided by Y.

REAL(X) - returns the real part of the complex number X.

ROUND(X,N) - returns X rounded to the nearest integer in N is 0. Returns X rounded to the Nth digit to the right of the decimal point if N>0. If N<0, X is rounded to the Nth digit to the left of the decimal point.

SIGN(X) - returns a fixed binary value equal to 1 if X>0, 0 if X=0, and -1 if X<0.

TRUNC(X) - returns the CEIL(X) if X<0 and FLOOR(X) if X>0.

ARRAY FUNCTIONS:

ALL(X) - returns a bit string obtained by 'and-ing' (&) all of the bit strings of array X together. X must be an array of bit strings.

ANY(X) - returns a bit string obtained by 'or-ing' (|) all of the bit strings of array X together. X must be an array of bit strings.

DIM(X,N) - returns the 'extent' of the Nth dimension of array X. The extent is the upper bound minus the lower bound plus one.

HBOUND(X,N) - returns the upper bound of the Nth dimension of array X.

LBOUND(X,N) - returns the lower bound of the Nth dimension of array X.

PROD(X) - returns the product of all the elements of array X.

SUM(X) - returns the sum of all the elements of array X.

MATHEMATICAL FUNCTIONS:

ATAN(X) - returns the arctangent of X.

ATAN(X,Y) - returns the arctangent of X/Y. X and Y must both be real and must not both be 0.

ATAN(X,Y) - returns the arctangent of X/Y in degrees. X and Y must be real and must not both be 0.

ATAN(X) - returns the hyperbolic tangent of X. The value of X must be greater than or equal to 0.

COS(X) - returns the cosine of X, where X is expressed in radians.

COSD(X) - returns the cosine of X, where X is expressed in degrees.

COSH(X) - returns the hyperbolic cosine of X.

EXP(X) - returns e**X where e is the base of the natural logarithm system.

LOG(X) - returns the natural logarithm of X. X should be greater than 0.

LOG10(X) - returns the common logarithm of X (base 10). X should be real and greater than 0.

LOG2(X) - returns the logarithm of X to base 2. X should be real and greater than 0.

SIN(X) - returns the sine of X, where X is expressed in radians.

SIND(X) - returns the sine of X, where X is expressed in degrees.

SINH(X) - returns the hyperbolic sine of X.

SQRT(X) - returns the square root of X. X should be real and greater than or equal to 0.

TAN(X) - returns the tangent of X, where X is expressed in radians.

TAND(X) - returns the tangent of X, where X is expressed in degrees.

TANH(X) - returns the hyperbolic tangent of X.

STRING FUNCTIONS:

HIGH(I) - returns a character string of length I, each character of which is the highest character in the collating sequence: hexadecimal FF.

INDEX(STRING, CONFIG) - returns the leftmost postion in STRING where the CONFIG begins. If CONFIG does not appear as a substring of STRING, a zero is returned. Both STRING and CONFIG must be bit or character strings.

LENGTH(STRING) - returns a number giving the length of the STRING,

LOW(I) - returns a character string of length I, each character of which is the lowest in the collating sequence: hexadecimal 00.

REPEAT(STRING,I) - returns the string concatenated with itself I times. Thus REPEAT('A',2) is 'AAA'. STRING should be a bit or character string and I a decimal constant.

STRING(X) - returns a string resulting from concatenating all elements of X together. X must be a variable, array name or structure composed entirely of bit or character strings.

SUBSTR(STRING,I,{,J}) - returns the substring of the STRING starting at character I, ending at character I+J-1. If J is missing, it is assumed to be LENGTH(STRING)-I+1.

TRANSLATE(S,R,P) - results in a string identical to S, except that any character of S which is in P is replaced by the corresponding character in R. For example

```
TRANSLATE('XYZW','ABCD','VWXY')
```

results in the string 'CDZB'.

UNSPEC(X) - results in a bit string containing the internal representation of X.

VERIFY(STRING,CONFIG) - returns the position of the first character in STRING that is not in CONFIG. If all characters from STRING are in CONFIG, it returns a zero.

TYPE CONVERSION FUNCTIONS:

BINARY(X) - returns X with a binary base.

DECIMAL(X) - returns X with a decimal base.

FIXED(X) - returns X converted to fixed point.

FLOAT(X) - returns X converted to floating point.

MISCELLANEOUS FUNCTIONS:

DATA - returns a character string of length 6, with the form YYMMDD where YY is the current year, MM is the current month, and DD is the current day.

LINENO(FILE_NAME) - returns the number of the current line on the named file.

TIME - returns a character string of length 9 giving the current time of day. Its form is HHMMSSTTT, where HH is the current hour of the day, MM is the number of minutes, SS is the number of seconds, and TTT is the number of milliseconds.

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-77-893	2.	3. Recipient's Accession No.

Title and Subtitle

AN INTRODUCTION TO PL/C

5. Report Date: September 21, 1977

6.

Author(s): Douglas D. Dankel, II

Performing Organization Name and Address
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

8. Performing Organization Rept. No. UIUCDCS-R-77-893

10. Project/Task/Work Unit No.

11. Contract/Grant No.

Sponsoring Organization Name and Address
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered

14.

Supplementary Notes

Abstracts

This report examines the PL/C programming language and basic programming techniques. It is intended for use as a supplemental text in introductory programming courses or for use by advanced undergraduate or graduate students unfamiliar with PL/C. Included are descriptions of the PL/C program statements, program format, arrays and structures, simple searching and sorting techniques, character string manipulation, procedures, advanced language features and programming techniques, and an explanation of PL/C error messages.

Key Words and Document Analysis. 17a. Descriptors

PL/C programming language, introductory text, programming techniques, searching, sorting, arrays, structures, error codes, error messages.

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	156
	20. Security Class (This Page) UNCLASSIFIED	22. Price